



User's Guide

VectorCAST/Interactive Tutorials

VectorCAST 2020

New editions of this guide incorporate all material added or changed since the previous edition. Update packages may be used between editions. The manual printing date changes when a new edition is printed. The contents and format of this manual are subject to change without notice.

Generated: 11/4/2020, 8:35 PM

Rev: 257ebc0

Part Number: VectorCAST Interactive Tutorials for VectorCAST 2020

VectorCAST is a trademark of Vector Informatik, GmbH

© Copyright 2020, Vector Informatik, GmbH All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any informational storage and retrieval system, without written permission from the copyright owner.

U.S. Government Restricted Rights

This computer software and related documentation are provided with Restricted Rights. Use, duplication or disclosure by the Government is subject to restrictions as set forth in the governing Rights in Technical Data and Computer Software clause of

DFARS 252.227-7015 (June 1995) and DFARS 227.7202-3(b).

Manufacturer is Vector Informatik, GmbH East Greenwich RI 02818, USA.

Vector Informatik reserves the right to make changes in specifications and other information contained in this document without prior notice. Contact Vector Informatik to determine whether such changes have been made.

Third-Party copyright notices are contained in the file: 3rdPartyLicenses.txt, located in the VectorCAST installation directory.

TABLE OF CONTENTS

Introduction	7
Starting VectorCAST	8
The Welcome Page	9
To Create the Tutorials Using the Welcome Page	10
Importing Examples into the Welcome Page	11
VectorCAST Overview	13
VectorCAST Automation	13
VectorCAST Components	14
VectorCAST/Cover Overview	15
Testing Overview	16
Key Terminology	16
Key Concepts	18
Tutorial Overview	19
Example Environments	20
After the Tutorials	21
C Tutorials	22
Introduction	23
Basic Tutorial	23
What You Will Accomplish	23
Preparing to Run the Tutorial	23
Building a Test Environment	24
Building Test Cases	39
Controlling Values Returned from Stubs	43
Adding Expected Results to the Test Case	45
Executing Test Cases	49
Generating Test Reports	51
Analyzing Code Coverage	52
Generating Coverage Reports	55
Using Stub-By-Function	57
Creating Regression Scripts	64
Tutorial Summary	65
Troubleshooting	65
Multiple UUT Tutorial	69
What You Will Accomplish	69
Building a Multiple UUT Environment	69
Building a Compound Test Case	72
Adding Test Iterations for Fuller Coverage	82
Adding Test Cases to a Compound Test	87
Exporting a Test Script	91
Tutorial Summary	91

C++ Tutorials	92
Introduction	93
Basic Tutorial	93
What You Will Accomplish	93
Preparing to Run the Tutorial	93
Building an Executable Test Environment	94
Building Test Cases	110
Controlling Values Returned from Stubs	115
Adding Expected Results to the Test Case	117
Executing Test Cases	123
Instantiating a Class Instance	123
Generating Test Reports	126
Analyzing Code Coverage	127
Generating and Managing Coverage Reports	131
Using Stub-By-Function	133
Creating Regression Scripts	140
Tutorial Summary	141
Troubleshooting	141
Multiple UUT Tutorial	145
What You Will Accomplish	145
Building a Multi UUT Environment	146
Building a Compound Test Case	149
Adding Test Iterations for Fuller Coverage	161
Adding Test Cases to a Compound Test	166
Exporting a Test Script	170
Tutorial Summary	170
Ada Tutorials	171
Introduction	172
Basic Tutorial	172
What You Will Accomplish	172
Preparing to Run the Tutorial	172
Building an Executable Test Environment	173
Building Test Cases	182
Controlling Values Returned from Stubs	188
Adding Expected Results to the Test Case	190
Executing Test Cases	194
Generating and Managing Test Reports	196
Analyzing Code Coverage	197
Generating and Managing Coverage Reports	200
Creating Regression Scripts	202
Tutorial Summary	204
Troubleshooting	204
Multiple UUT Tutorial	207
What You Will Accomplish	207

Building a Multi UUT Environment	208
Building a Compound Test Case	211
Adding Test Iterations for Fuller Coverage	220
Adding Test Cases to a Compound Test Case	225
Exporting a Test Script	229
Tutorial Summary	229
Whitebox Tutorial	230
What You Will Accomplish	231
Building a Whitebox Test Environment	231
Specifying a Working Directory	231
Specifying the Type of Environment to Build	232
Specifying the source files to be tested	233
Designating UUTs and stubs	234
Compiling your specifications into an executable test harness	235
Creating a Test Case for a Hidden Subprogram	236
Tutorial Summary	238
Cover Tutorials	239
Introduction	240
C++ Cover Tutorial	240
What You Will Accomplish	240
Creating a VectorCAST/Cover Environment	241
Building the Tutorial Application	244
Executing the Instrumented Program	247
Adding Test Results	248
View the Aggregate Coverage Report	251
View the Management Report	252
Tutorial Summary	255
Ada Cover Tutorial	255
What You Will Accomplish	256
Creating a VectorCAST/Cover Environment	256
Building the Tutorial Application	260
Executing the Instrumented Program	263
Adding Test Results	264
View the Aggregate Coverage Report	266
View the Management Report	267
Tutorial Summary	270
Integration Tutorials	272
Introduction	273
Integrating Lint	273
Preparing to Run the Tutorial	273
Build the Tutorial Environment	273
Using Lint to Analyze Source Code	275
Using VectorCAST/Lint's MISRA Compliance Checking	279

Homework	282
Tutorial Summary	282
Integrating PRQA	283
Preparing to Run the Tutorial	283
Using QA•C++ to Analyze Source Code	283
Tutorial Summary	290
Enterprise Testing Tutorial	291
 Introduction	292
How Enterprise Testing Works	292
Terminology	292
Creating a VectorCAST Project	293
What You Will Accomplish	293
Preparing to Run the Tutorial	293
Using the VectorCAST Project Wizard	295
The Project Tree	300
The Status Panel	301
Building and Executing the Environments	302
Modifying the Project	305
Tutorial Summary	310
 Appendix A: Tutorial Source Code Listings	311
Requirements	311
C	313
C++	318
Ada	326
Build Settings	334
Index	340

Introduction

Starting VectorCAST

Prior to starting VectorCAST, you should ensure that VectorCAST is installed. See the *VectorCAST Installation Guide* for installation details. Also, ensure that the environment variable `VECTORCAST_DIR` is set to the installation directory.

UNIX

In UNIX or Cygwin, you start VectorCAST by entering:

```
$VECTORCAST_DIR/vcastqt
```

The compiler you are using must be included in your PATH environment variable. If it is not included, modify your PATH environment variable with the set, setenv, or export command (as appropriate to your shell), and then start VectorCAST from the same shell window.

Windows

In Windows, you can start VectorCAST from the Start menu or from a command line.

From the command line (DOS command prompt), enter:

```
%VECTORCAST_DIR%\vcastqt
```

From the Windows Start menu, select **Start => All Programs => VectorCAST => VectorCAST**.

The compiler executables must be found in a directory specified on your `PATH` environment variable. You can access your `PATH` environment variable by selecting the **System** icon on the Windows Control Panel.

If you start VectorCAST without specifying the location of the license file, a FLEXIm License Finder dialog appears.



This dialog enables you to specify the path to the license file or the `port@host` from which the

VectorCAST license is served. When you click **OK**, the information entered is saved in the Windows registry at location:

```
HKEY_LOCAL_MACHINE (HKEY_CURRENT_USER on Windows7 and Vista)
  SOFTWARE
    FLEXIm License Manager
      VECTOR_LICENSE_FILE
```

Thereafter, VectorCAST will use this information when starting.



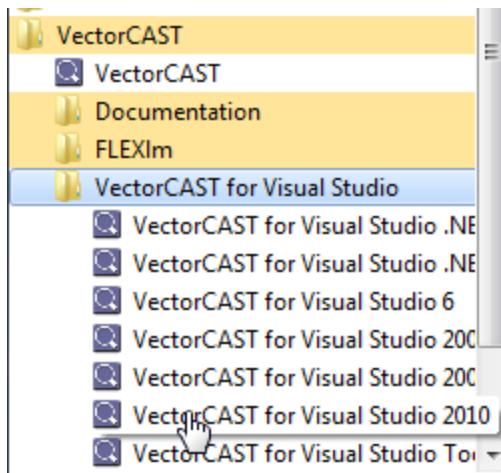
Note: If you do not want to modify the Windows registry, set the environment variable **FLEXIM_NO_CKOUT_INSTALL_LIC** to 1.

If FLEXIm cannot find **VECTOR_LICENSE_FILE** in the Windows registry, it looks for **LM_LICENSE_FILE**. If that is not defined in the registry, FLEXIm uses environment variables **VECTOR_LICENSE_FILE** and **LM_LICENSE_FILE**, in that order.



Note: If you do not want FLEXIm to read the Windows registry to find the license file, set the environment variable **LM_APP_DISABLE_CACHE_READ** to 1. Note that this environment variable controls registry reading for all FLEXIm applications, not just VectorCAST.

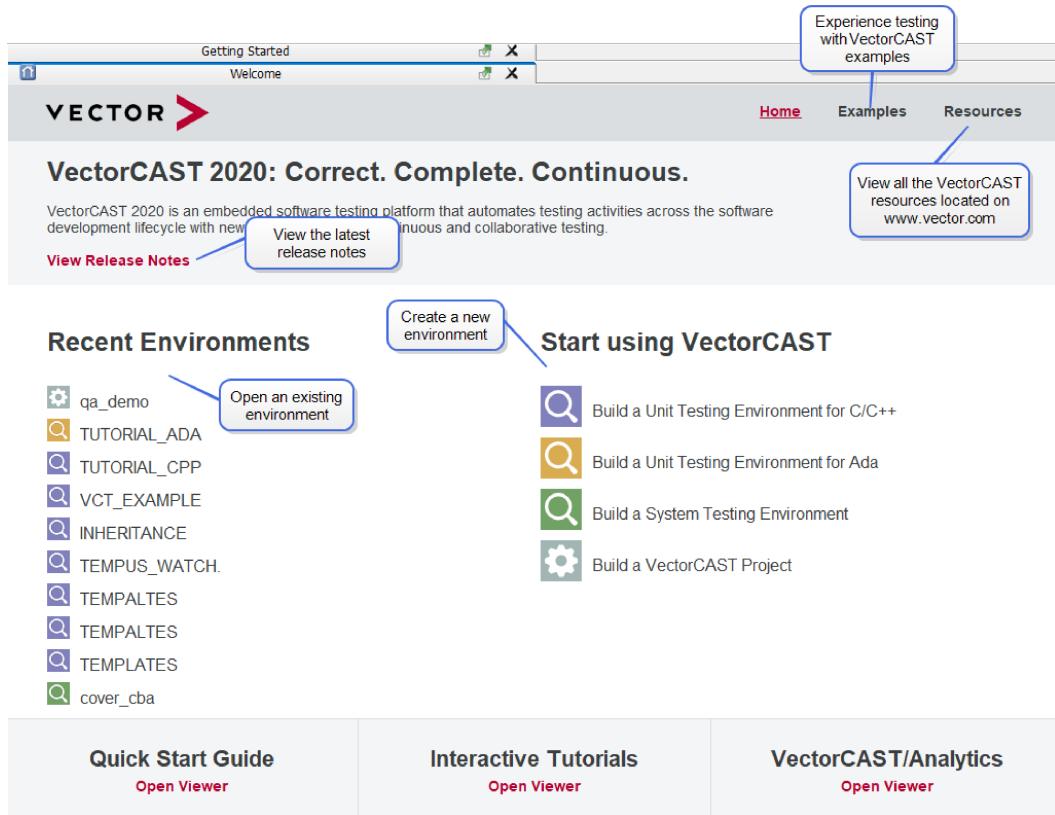
To use one of the Microsoft Visual Studio compilers, select **Start menu => All Programs => VectorCAST => VectorCAST for Visual Studio**. Then select one of the compiler-version options under **VectorCAST for Visual Studio**.



When you select a **VectorCAST for Visual Studio** option, the **vcvars32.bat** or **vsvars32.bat** file automatically runs to ensure that the associated compiler is included in your PATH environment variable.

The Welcome Page

After starting VectorCAST with either the **vcastqt** command or the Windows Start menu, the VectorCAST Welcome page appears.



On this screen you can choose to:

> **Start using VectorCAST**

Create a new or open an existing test environment.

> **Open the Quick Start Guide**

Use this quick reference guide to get started with the basic features of VectorCAST.

> **Use the Interactive Tutorials**

Tutorials are provided for each development language supported by VectorCAST (C, C++, and Ada), which introduce the basic commands and demonstrate multiple-unit support. Tutorials are also provided for the VectorCAST/Cover and Enterprise Testing tools.

> **VectorCAST/Analytics**

Learn about VectorCAST/Analytic's easy-to-understand web-based dashboard view of software code quality and testing completeness metrics. Analytics enables users to identify trends in a single codebase or compare metrics between multiple codebases.

> **Read what's new with VectorCAST**

Access release notes for the current major release, read the latest news about testing, and press releases about VectorCAST and join the VectorCAST community.

To Create the Tutorials Using the Welcome Page

Select the **Examples** link from the Welcome page navigation bar in the upper right.

The VectorCAST examples allow you to easily experience the ease of testing with VectorCAST.

Each Unit Test example consists of easy to understand source code and the required VectorCAST scripts to build and run some test for that code. When you select an example, VectorCAST will automatically build the test environment, load the example test cases, and display an HTML overview of the tool features demonstrated by the example.

The examples use the MinGW compiler that is bundled with VectorCAST, so no compiler configuration is required.

The Enterprise Unit Testing example will build a VectorCAST Project using Unit Testing Examples.

All of these examples can be accessed from the **Help > Examples** menu.

C Unit Testing

- Tutorial for C
- Stubbing malloc
- CSV-based Testing
- Testing void* Parameters

C++ Unit Testing

- Tutorial for C++
- Basic Class (BlackBox)
- Basic Class (Whitebox)
- Class Inheritance
- Namespaces
- STL Containers
- C++ Templates
- C++ Exceptions

Enterprise Unit Testing

- Enterprise Unit Testing Example

Ada Unit Testing

- Tutorial for Ada

System Testing

- System Testing Example
- Google Test Example
- CppUnit Example

Static Analysis

- Static Analysis Example

Change destination: C:\VCAST\examples\environments
Hyperlinks are only available for licensed products.
[Reset Examples](#)

Select an example or tutorial and VectorCAST will automatically build the test environment, load the example test cases and display an HTML overview of the tool features demonstrated in the example.

The examples are also accessible through **Help => Example Environments**.

Importing Examples into the Welcome Page

Any VectorCAST environment or VectorCAST project can be imported to as part of the list of available examples. After importing, the item appears both on the Welcome page and in the **Help => Example Environments** context menu.

Examples are imported from a .csv file as described below. For illustration the following file will be created:

```
%VECTORCAST_DIR%\Examples\MyExamples.csv
```

This file contains a single line entry for each imported example.

Each entry in the .csv is defined as follows:

- > Column 1 specifies the language used for a VectorCAST environment or a dash ("") for a VectorCAST Project
 - >> Ada
 - >> C
 - >> C++
 - >> -

- > Column 2 specifies the example type:
 - >> UnitTest
 - >> Cover
 - >> Manage
 - >> QA
 - >> Lint
 - > Column 3 specifies the relative path to the example's .env or shell script. This path is relative to the \$VECTORCAST_DIR/Examples directory.
 - > Column 4 specifies a prefix to be appended to the README.html file (if one is available) when the environment is built in %VECTORCAST_DIR\Examples\environments
 - > Column 5 specifies the category (Ada, C, C++, Enterprise Testing, System Testing, or Static Analysis) for the example and the name of the example as it will appear on the Welcome page and in the **Help => Environments** context menu.
- >> The format for this column is "CATEGORY -> NAME".

To illustrate, the following line is entered into MyExamples.csv":

```
C, UnitTest, c/stubbing_c_lib/C_LIB_STUB.env, projectDemo, "C -> Example Demo"
```

After the changes are saved to MyExamples.csv, select **Help => Import Examples** ... and using the file chooser, select MyExamples.csv..

If the Welcome screen is currently open, close it. Open the Welcome screen by selecting **Help => Welcome** and then select **Examples** from the navigation menu in the upper right. The new example, called Example Demo, appears as a choice in the C Unit Testing column.

The screenshot shows the VectorCAST Welcome screen with the 'Examples' section selected. The 'C Unit Testing' category is highlighted, showing examples like 'Tutorial for C', 'Stubbing malloc', 'CSV-based Testing', 'Testing void* Parameters', and 'Example Demo (i)'. A blue callout box with the text 'New example imported into the list' points to the 'Example Demo (i)' entry. Other categories shown include 'C++ Unit Testing', 'Enterprise Unit Testing', 'System Testing', and 'Static Analysis'. Each category has its own set of examples listed below it. At the bottom, there is a note about change destination and a 'Reset Examples' button.

Examples

The VectorCAST examples allow you to easily experience the ease of testing with VectorCAST.

Each Unit Test example consists of easy to understand source code and the required VectorCAST scripts to build and run some test for that code. When you select an example, VectorCAST will automatically build the test environment, load the example test cases, and display an HTML overview of the tool features demonstrated by the example.

The examples use the MinGW compiler that is bundled with VectorCAST, so no compiler configuration is required.

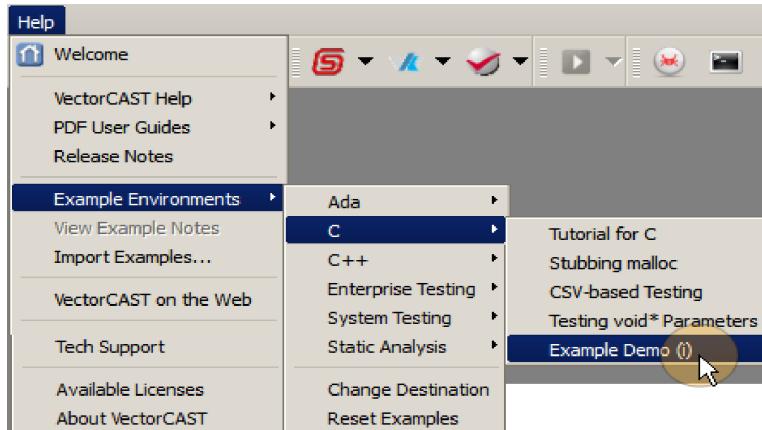
The Enterprise Unit Testing example will build a VectorCAST Project using Unit Testing Examples.

All of these examples can be accessed from the **Help > Examples** menu.

C Unit Testing	C++ Unit Testing	Enterprise Unit Testing
Tutorial for C Stubbing malloc CSV-based Testing Testing void* Parameters Example Demo (i)	Tutorial for C++ Basic Class (BlackBox) Basic Class (Whitebox) Class Inheritance Namespaces STL Containers C++ Templates C++ Exceptions	Enterprise Unit Testing Example
Ada Unit Testing	System Testing	
Tutorial for Ada	System Testing Example Google Test Example CppUnit Example	
	Static Analysis	
	Static Analysis Example	

Change destination: C:\VCAST\examples\environments
Hyperlinks are only available for licensed products.
[Reset Examples](#)

Select **Help => Example Environments => C**, and the new “Example Demo” entry is available on the context menu.



VectorCAST Overview

VectorCAST® is a suite of tools for automating the entire process associated with conducting unit and integration testing:

- > VectorCAST/C++™ and VectorCAST/Ada™ (collectively VectorCAST) automatically generate executable test harnesses for one or more units of application source code written in C/C++ or Ada/Ada95. In addition, they generate and execute test cases and report on results.
- > VectorCAST/Cover is a code-coverage analysis tool that identifies which areas of an application have been exercised by test executions. It supports Ada, C, and C++ modules in a multi-language environment, and supplements the unit coverage provided by VectorCAST/C++ and VectorCAST/Ada.

This *Interactive Tutorials Guide* focuses on VectorCAST/C++, VectorCAST/Ada, and VectorCAST/Cover.

VectorCAST Automation

You can use VectorCAST to conduct unit or integration testing on applications written in the C, C++, or Ada programming languages. VectorCAST automates the following unit and integration test functions:

- > Generating and compiling test stubs and driver programs
- > Generating test cases based on boundary values
- > Constructing user-defined test cases by interactive point-and-click, or by script
- > Executing modified test cases without re-compiling the test harness (Exception: when running in stdout only mode, the test harness must be recompiled.)
- > Conducting regression testing
- > Generating standards-compliant test reports
- > Conducting comprehensive branch and statement coverage analysis
- > Conducting Modified Condition/Decision Coverage (MC/DC) analysis

- > Conducting basis-path analysis and determining cyclomatic complexity
- > Executing tests on host and embedded-target development systems

Three types of input data can affect the processing of a unit under test (UUT):

- > The input parameters to the UUT
- > The output parameters of dependent-unit subprograms (that is, the output of subprograms invoked by the unit)
- > Global data objects

VectorCAST provides direct control over all three types of data. It does this by automatically creating a unique test harness for each software unit to be tested.

VectorCAST Components

VectorCAST consists of four major functional components:

- > Test Environment Builder
- > Test Case Generator
- > Execution Manager
- > Report Generator

Test Environment Builder

The Test Environment Builder is central to VectorCAST and serves two related functions:

- > It generates a framework (environment) consisting of all the necessary resources to build a test harness
- > It generates (or regenerates) the test harness itself

To initiate a test environment, the user:

- > Supplies the location of the source files to undergo test
- > Designates the individual UUT(s) to be exercised
- > Designates any dependent units to be stubbed

The Environment Builder parses the designated files and, from the information gathered, generates a complete test harness.

Unlike a manually generated test harness, a VectorCAST test harness can be automatically regenerated whenever changes are made to a UUT or stub included in the environment (for example, when a subprogram is fully coded).

In addition, test cases can be added to an existing environment without any need for manual coding, recompiling, etc.

Test Case Generator

The Test Case Generator uses static analysis and/or user input to build test cases for exercising the UUTs included in a test environment. These test cases (and their results) are stored within the test environment itself and are thereby available for use in future testing. This is an important feature for

regression testing.

Execution Manager

The Execution Manager invokes the test harness created by the Environment Builder, and executes the test cases built by the Test Case Generator.

Report Generator

The Report Generator produces a variety of reports based on information maintained in every environment database. This information includes:

- Test configurations** – Unit names, test names, dates and times.
- Test case data** – Listings of inputs and expected results.
- Execution histories** – Listings of execution results; comparisons of expected and actual results; comparisons of control flow; pass/fail indications.
- Coverage** – Source listings annotated and color-coded to indicate code coverage.
- Results** – Tables summarizing pass/fail status and coverage results.

VectorCAST/Cover Overview

VectorCAST/Cover is a code-coverage analysis tool that identifies which areas of an application have been exercised by test executions. It supports Ada, C, and C++ modules in the same execution environment, and supplements the unit coverage provided by VectorCAST/C++ and VectorCAST/Ada.

You can apply VectorCAST/Cover to particular parts of an application or to the entire application at once. In addition, you can also import coverage data from VectorCAST, which includes a utility for determining the coverage on one or more units.

VectorCAST/Cover adds instrumentation statements to the source files to be tested and analyzed. These statements capture information about code coverage at the time of execution. You access this information by generating reports based on a single execution or on multiple executions.

VectorCAST/Cover supports statement, branch/decision, and industry-specific statement + branch coverage as a standard feature. It also supports Modified Condition/Decision Coverage (MC/DC), and industry-specific statement + MC/DC coverage, under separate licensing.

In addition to code coverage, VectorCAST/Cover supports code-complexity analysis and basis-path analysis. Code-complexity analysis determines the number of unique paths through a function or subprogram. Basis-path analysis determines the values required at each decision point to exercise all unique paths through a unit of code.

Using VectorCAST/Cover involves the following general procedure:

1. Building a coverage environment
 - > Creating an environment in VectorCAST/Cover, and including in this environment the application source files to be tested and analyzed.
 - > Specifying the type of instrumentation to be applied to the source files.

2. Building a test application
 - > Compiling the instrumented source files (as well as any other files required by the application).
 - > Compiling the I/O library files provided with VectorCAST/Cover.

These files enable VectorCAST/Cover to report on the source statements exercised during program execution. (You perform this step outside VectorCAST/Cover.)

 - > Linking the compiled (object) files into an executable program.
3. Generating test results
 - > Using your system/functional test cases to exercise the application.

As the application executes, coverage data is captured and written to a data file (named TESTINSS.DAT).

 - > Importing coverage data from other VectorCAST environments.

You can import coverage results from any VectorCAST/C++ or VectorCAST/Ada environment. Similarly, you can import coverage results from a VectorCAST/Cover environment into a VectorCAST/C++ or VectorCAST/Ada environment.
4. Viewing coverage results
 - > Generating/displaying/printing coverage reports.

You can tailor the content and formatting of any coverage report to meet your specific needs.

Testing Overview

This section presents some terms and concepts you should be familiar with before you begin using the VectorCAST tutorials.

Key Terminology

You should be familiar with the following terms before you begin using the tutorials:

Environment – The term “Environment”, as it is used in this document, refers to a repository of information saved in a VectorCAST-created disk directory. This directory is used to store test harness source files, test-specific data files, etc.

Test Harness – A test harness is the executable program created when the test driver, the units under test, and the dependent units or stubs of dependent units are linked together.

The test harness has three components:

> **Units Under Test (UUTs)**

A Unit Under Test is a compilation unit to be tested. In C or C++, a UUT is a single source file. In Ada, it can be a standalone subprogram specification, subprogram body, a package specification, or package body. You can have more than one unit under test in a test environment (for integration testing).

> **Test Driver**

The driver program is capable of invoking all subprograms of the unit under test with values provided from the test case data and capturing any returned data or exceptions that may be raised during test execution. The components of the test harness are automatically compiled and linked into an executable test harness, which is used by VectorCAST to

effect all testing.

> Smart Stubs

When testing a software unit, it is often desirable to create alternate subprogram definitions for some or all dependent units. These “placeholders” used for testing purposes are known as *stubs*. Stubbed subprograms allow you to begin testing long before the actual system is built. They also allow you to exercise more exact control over the values passed back to the Unit Under Test than would be available with the actual dependent unit in place.

VectorCAST automatically generates the code for those dependent units chosen to be stubbed.

Visible Subprogram – In C, a visible subprogram is a subprogram that is not declared as “static”; in C++, it is a member function that is not declared “private” or “protected;” in Ada, it is a subprogram specification contained in the visible part of a compilation unit.

Dependent Unit – A subprogram that contains a function called by a UUT, or data objects used by a UUT. A dependent unit is either a C/C++ file that is included or an Ada package that is ‘withed’ by a unit. Dependent units can be stubbed or used ‘as is’.

Event – An event is a change in the flow of control that VectorCAST can monitor. Each of the following is an event:

- > A call from the test harness to a UUT
- > A call from a UUT to a stubbed dependent in the test harness
- > A return from a UUT to the test harness

Test Case – A collection of input and output data, defined with the purpose of exercising specific software and verifying results. This data is commonly comprised of the formal input and output parameters of the dependent subprograms, and input and output parameters of the subprogram under test. The Test Data controls the execution of the unit under test as well as values for global objects visible in the unit under test or stubbed subprograms.

Test Driver – VectorCAST generates the main program, or *driver*, necessary to exercise all visible subprograms in all units under test. This consists of calls to each visible subprogram in a unit under test.

Search List (C/C++) / Parent Library (Ada) – The directory that contains the units under test and dependent units. In C or C++ environments, the Search List can contain multiple directories; in Ada environments, the Parent Library is a single directory in the hierarchy of libraries.

Prototype Stubbing – In C and C++ environments, VectorCAST can generate stubs of dependent units using only the function’s declaration. If VectorCAST cannot find the function’s definition (the implementation of the function) in any C/C++ source file in the Search List, then it creates a stub of the function using the function declaration (from a header file). Prototype-stubs are grouped under the name **Stubbed Subprograms** instead of under the name of a unit.

Stub by Function Units – Prior to VectorCAST version 5.0, only subprograms in dependent units of the UUT were allowed to be stubbed. In fact, all subprograms in stubbed units had to be stubbed. In VectorCAST 5.0, you have the choice of making a Unit Under Test “stub by function (SBF).” When a UUT is made “SBF”, this means that functions in the unit can be stubbed individually, while others in the unit are not stubbed.

Testable Class – In C++ environments, a testable class is one that has any member function defined in a UUT source file, or an inlined function defined in a header that is included by a UUT source file.

Key Concepts

You should be familiar with the following concepts before you begin using the tutorials:

Execution Closure – An execution closure consists of all the units required for an application to successfully link. [Figure 1](#) shows a closure consisting of a single UUT, three direct dependent units, and several sub-dependents.

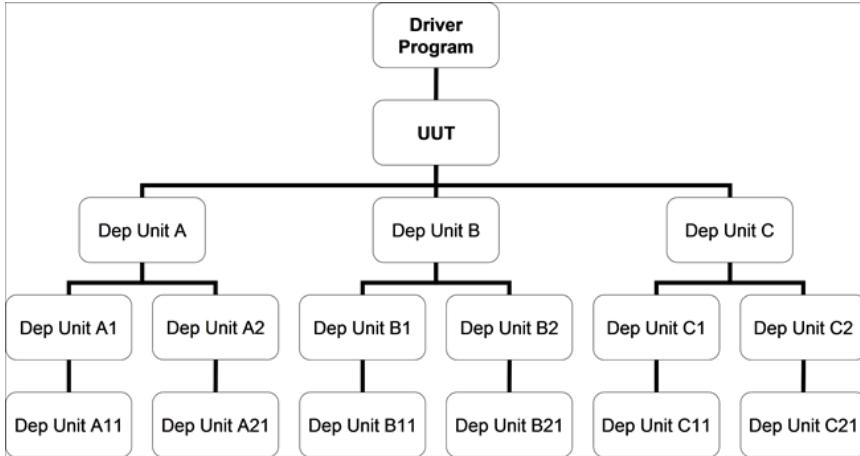


Figure 1. An example execution closure.

Stubbing Units in a Closure – When building a VectorCAST environment, you designate stubbing from the top-most level of dependency downward to the bottom-most level in each branch.

In the closure illustrated in [Figure 1](#), for example, dependents Dep Unit A, Dep Unit B, and Dep Unit C would be the primary candidates for stubbing. If you were to leave one of these dependents unstubbed, the dependents immediately under it would then become the primary candidates for stubbing, and so on to the bottom of each branch.

[Figure 2](#) shows the example closure after stubbing has been designated.

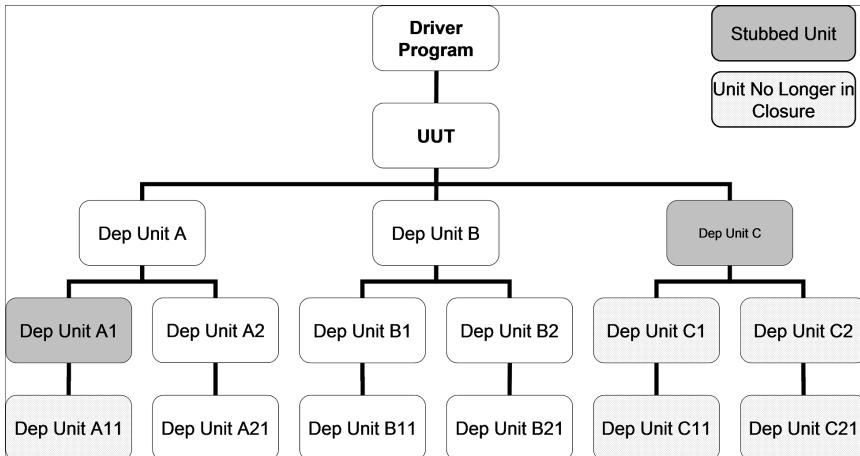


Figure 2. A typical VectorCAST closure.

The scheme allows a flexibility spanning from testing a single unit in a standalone environment in which

every dependent is stubbed, to testing groups of units with only the dependents on the periphery (bottom) of the closure stubbed.

Ignoring Units in a Closure (Ada only) – As mentioned above, normally when you choose not to stub a unit, any dependents below this unit remain in the closure. In VectorCAST/Ada, however, you can choose to leave a unit unstubbed but effectively remove all the downstream dependents from the list of stub candidates; all downstream dependents will not be stubbed. You do this by directing VectorCAST/Ada to ignore the unit.



Note: In Ada, ignoring dependents applies only to the dependents of the 'body' of the ignored unit; it does not apply to the dependents of the 'spec'. The reason for this is that VectorCAST needs to parse the dependents of the 'spec' in order to resolve types derived from the dependents.

This option is useful when you want to avoid doing any stubbing beyond a certain point. For example, in [Figure 3](#), suppose Unit C is an interface to a subsystem already tested. There is no need to stub beyond Unit C. By directing VectorCAST to ignore Unit C, it will use this entire branch as if you had chosen not to stub the units.

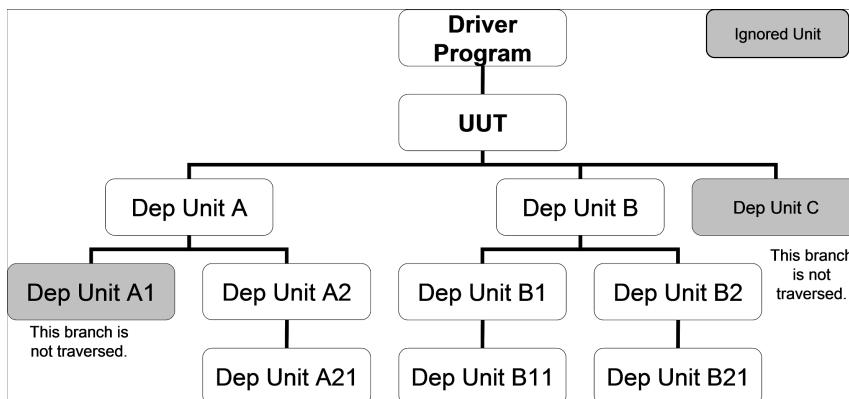


Figure 3. Closure with ignored units (Ada only)

This option can in some cases dramatically reduce the time it takes for VectorCAST to build an environment.



Note: In choosing to ignore a unit, downstream dependents will not be ignored if these units are brought into the closure through an alternate dependency.

Tutorial Overview

VectorCAST was designed to be analogous to the work you currently perform manually. This work normally includes building the test code (stubs and drivers) to invoke the unit under test and simulate dependent program interfaces, defining and building test cases to perform a particular function, executing the test cases, and collecting data from the test case execution to build the necessary reports.

Tutorials are provided for each development language supported by VectorCAST, organized into chapters by language (C, C++, and Ada). The first tutorial (Basic) in each chapter serves to introduce the most commonly used commands; the second (Multiple UUT) demonstrates VectorCAST's

multiple-unit support.

Tutorials are also provided for the VectorCAST/Cover tool, and for the VectorCAST build settings repository.

The tutorials use the source files listed in Table 1. Source Files Used in the Tutorials. You will find the complete contents of these files in "Tutorial Source Code Listings" on page 311.

Table 4. Source Files Used in the Tutorial

C Source Files	C++ Source Files	Ada Source Files	Purpose
ctypes.h	cpptypes.h	types.ads	Header file
database.c	database.cpp	database.adb	Dependent unit
manager.c	manager.cpp	manager.adb	UUT
	database.h	database.ads	Header file
	manager.h	manager.ads	Header file
manager_driver.c	manager_driver.cpp	manager_driver.adb	Driver program for manager and database
whitebox.c	whitebox.cpp	whitebox.adb	Whitebox example
	whitebox.h	whitebox.ads	Header file
place_order.tab			Tab-separated test data file
	Makefile		Makefile for Wrap_compile" tutorial

All the source files listed above are included in the VectorCAST installation subdirectory named **Tutorial**.

Example Environments

Example environments are included as part of the Welcome screen, see "To Create the Tutorials Using the Welcome Page" on page 10.

Also included in the VectorCAST installation directory is a subdirectory (Examples) that contains environments that demonstrate, in the case of VectorCAST/C++ and VectorCAST/Ada, how these tools handle certain language-specific concepts (inheritance and class, for example); and, in the case of VectorCAST/Cover, the different types of coverage this tool supports (statement, branch, etc.).

The directory for each example contains a Readme file. This file describes the concept or feature being demonstrated.

To build an example environment:

1. Start VectorCAST.
2. (Optional) Set the working directory to the directory containing the **.env** file you want to run.
For example, select **File => Set Working Directory**, and then **c:\vcast\Examples\cpp\inheritance**. Click **OK**.



Note: In UNIX, copy the example files to a location on your local machine, and then set the working directory to this location.

3. Select **File => New => C/C++ Environment**.
4. In the Create New Environment wizard, select the **Choose Compiler** option, and then select a C++ compiler from either drop-down menu.
5. Click the **Load** button.
6. Select the **.env** file showing in the **Choose File** dialog (e.g., **SHAPES .env**).



Tip: In the **File Open** dialog, ensure that the **Look In** field contains the correct subdirectory (for example, **inheritance**).

7. Click **Open**.
An environment name (e.g., **SHAPE**) appears in the Environment Name field.
8. Click the **Build** button (bottom far right).



Note: If the Build button is not enabled, you will need to make corrections before you can build your environment.

Once the environment is built, import the test script that is included in the example directory: **Test => Scripting => Import Script**.

For each test case, click the **Notes** tab to learn what this test case is demonstrating. To access the **Notes** tab, double-click on the name of the relevant test case in the Environment View.

9. Execute the test cases: **Test => Execute Batch All**.

After the Tutorials

When you have completed the tutorials relevant to your needs, you might want to use the environments you built to familiarize yourself with other VectorCAST capabilities, including:

- > Exporting/importing test scripts
- > Viewing other types of reports
- > Stepping through coverage with VectorCAST animation facility
- > Viewing code complexity and basis-path analysis data
- > Creating regression scripts
- > Using regression scripts to invoke VectorCAST functionality from CLICAST (Command Line Interface to VectorCAST)

C Tutorials

Introduction

The tutorials in this chapter demonstrate how to use VectorCAST/C++ to unit-test an application written in the C programming language.

For a comprehensive explanation of the VectorCAST/C++ features, refer to the *VectorCAST/C++ User's Guide*.

The modules you will test in the first two tutorials are components of a simple order-management application for restaurants. The source listings for this application are available in Appendix A, "Tutorial Source Code Listings" on page 311. It is recommended that you at least scan through these listings before proceeding with the tutorials.

You should run the tutorials in this chapter in the order presented.



Tip: You can stop a tutorial at any point and return later to where you left off. Each tutorial session is automatically saved. To return to where you left off, simply restart VectorCAST and the use **File => Recent Environments** to reopen the tutorial environment.

Basic Tutorial

The Basic Tutorial will take you through the entire process of using VectorCAST/C++ to test a software unit.

What You Will Accomplish

In this tutorial, you will:

- > Build a test environment (which includes an executable test harness)
- > Create test cases for a unit under test (UUT)



Tip: You can build the test environment and create the test cases simply and easily using the Welcome Screen, "To Create the Tutorials Using the Welcome Page" on page 10.

- > Execute tests on a UUT
- > Analyze code coverage on a UUT
- > Stub a function in the UUT

Please be aware that this tutorial does not provide a comprehensive explanation of VectorCAST's features. To gain a complete understanding of VectorCAST, we recommend that you read *VectorCAST/C++ User's Guide* before attempting to test your own code.



Note: User entries and selections are represented in this tutorial in bold. For example: "Enter **Demo**, then click **OK**".

Preparing to Run the Tutorial

Before you can run this tutorial, you need to decide on a working directory and copy into this directory the Tutorial files from the VectorCAST installation directory. On Windows, a good choice is the default

working directory, named **Environments**, located in the VectorCAST installation directory. On UNIX, it often is desirable to create your own working directory.

For efficiency, this tutorial assumes a working directory named **vcast_tutorial** located at the top level in your file tree, and a sub-directory named **c**. You can, however, locate this directory any place you want, and make the necessary translations in the text as you progress through the tutorial. Also for efficiency, this tutorial assumes you will be running the tutorial in a Windows environment, using Windows syntax for file paths and things specific to an operating system (Windows, Solaris, and UNIX). Again, you should have no problems making the necessary translations.

The files you need to copy are located in the directory in which you installed VectorCAST. On Windows, these files are located in **%VECTORCAST_DIR%\Tutorial\c** by default.

In Windows, enter:

```
C:\>mkdir vcast_tutorial\c  
C:\>copy %VECTORCAST_DIR%\Tutorial\c vcast_tutorial\c
```

In UNIX, enter:

```
$ mkdir vcast_tutorial/C  
$ cp $VECTORCAST_DIR/tutorial/cpp/* vcast_tutorial/C
```

Building a Test Environment

In this section, you will use the VectorCAST/C++ tool to create an executable test environment. The sample UUT you will use is named **manager**.

You will:

- > Specify a working directory
- > Specify an environment type (C/C++)
- > Specify a compiler
- > Name the environment to be created
- > Turn on whitebox
- > Specify the source files to be included in the environment
- > Designate the UUT and any stubs (as well as the method of stubbing)
- > View a summary of your specifications
- > Build the test environment

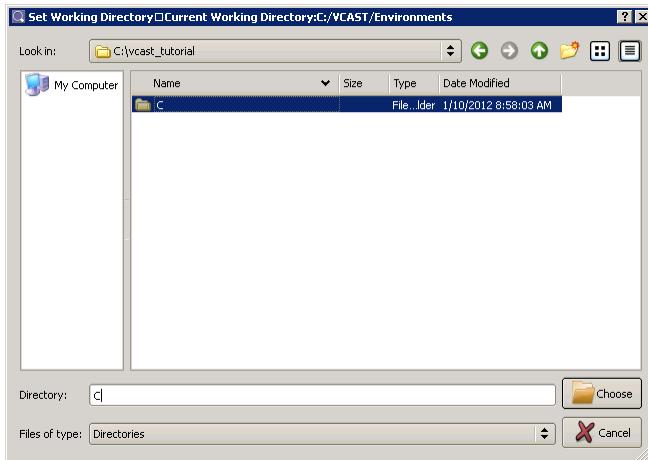
Specifying a Working Directory

The working directory contains the files you will use to build your test harness. For the purposes of this tutorial, the working directory is **vcast_tutorial\C**

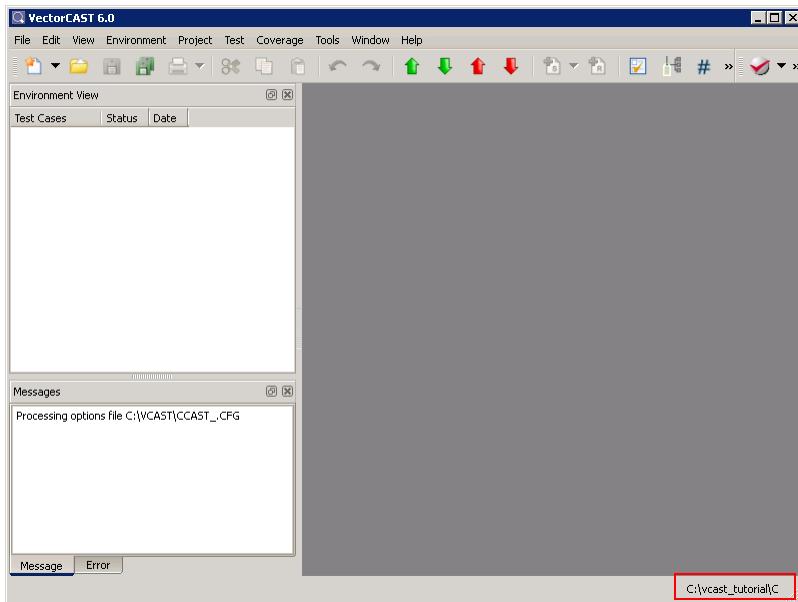
For details on starting VectorCAST, refer to "Introduction" on page 7.

1. With the VectorCAST main window open, select **File => Set Working Directory**.
A browsing dialog appears.

2. Navigate to **C:\vcast_tutorial\C** ; click **Choose**.



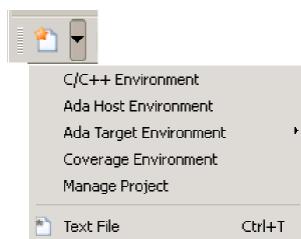
The location and name of the working directory you specified appear at the lower right-hand corner of the VectorCAST main window:



Specifying the Type of Environment to Build

1. Click the **New** button  on the toolbar.

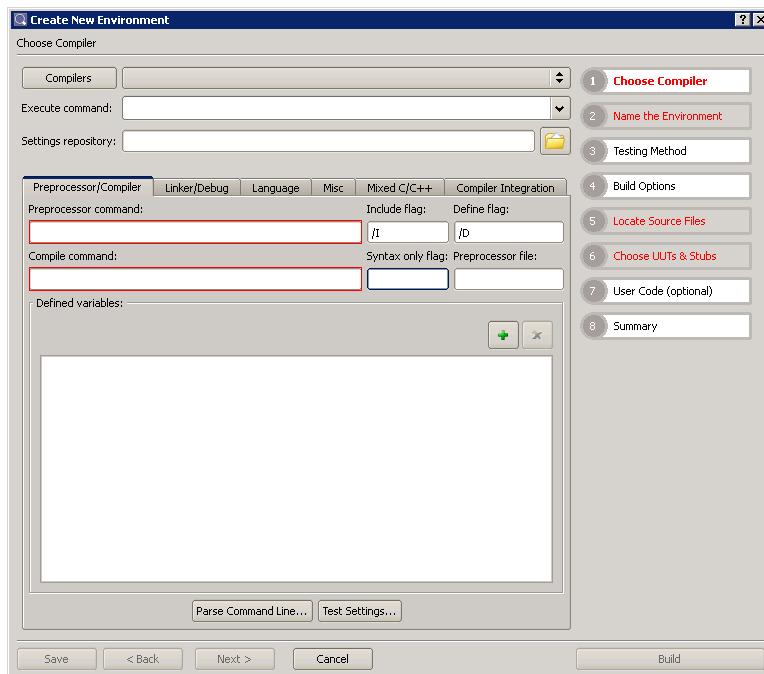
If more than one VectorCAST product is licensed, a drop-down menu appears listing the available environment types:



If the only type of environment licensed is C/C++, the Create New Environment wizard appears when you click the New button.

2. Select **C/C++ Environment**.

The Choose Compiler page of the Create New Environment wizard appears:



Selecting a Compiler

You use the Choose Compiler page to specify a compiler. The compiler you specify must be included in your PATH environment variable.

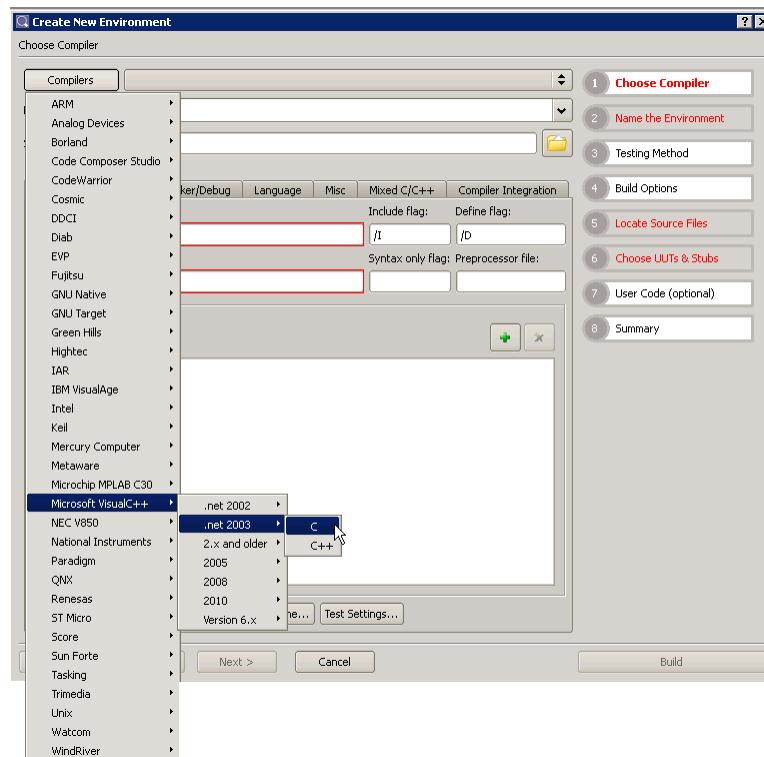
- To access a list of compilers supported by VectorCAST, click the down arrow on the Compilers button. Select a C compiler from the dropdown menu.

In this tutorial, you are building a C environment; therefore, you must select a C VectorCAST compiler template. Most compiler templates have both C and C++ variants.

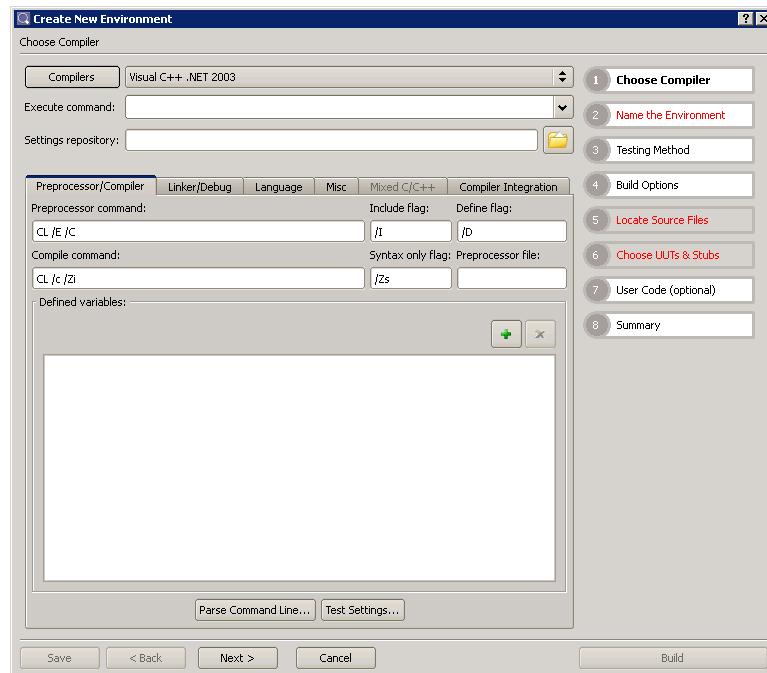


Tip: If you are planning to use VectorCAST with a target compiler, it is recommended you first go through the tutorials using a Host version of your compiler. Once you are familiar with the basic functionality of VectorCAST, refer to the VectorCAST/RSP User's Guide for information specific to your target.

For example, if you are using the Visual C++ .NET 2003 compiler, you would select from the cascading menus as shown here



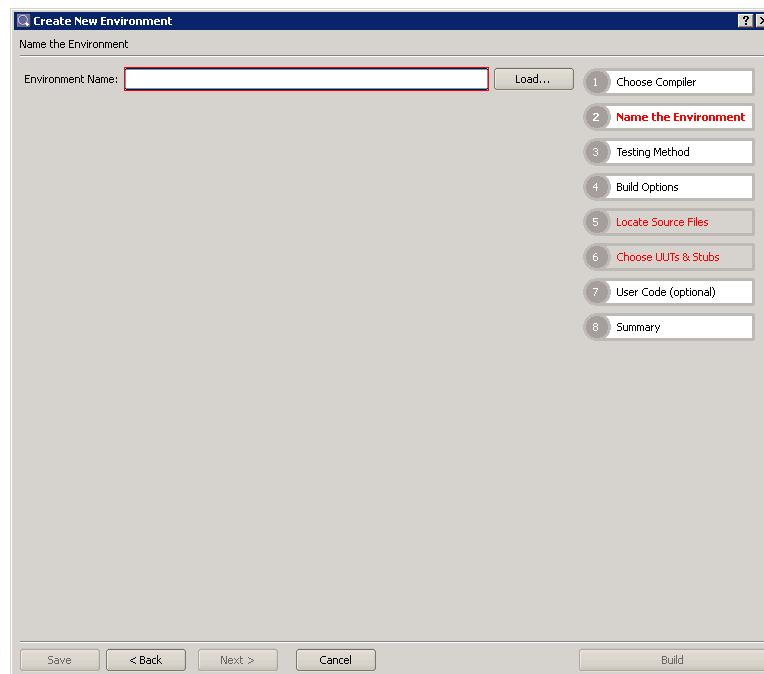
After you select a compiler, VectorCAST fills in the Preprocessor command and Compile command fields with the correct commands.



If VectorCAST cannot find these commands on your default path, it outlines the fields with a red border. In this case, hover your cursor over either field and a tool tip will explain the problem; then click **Cancel**, exit VectorCAST, and address the problem. (Most likely, the compiler you specified is not included in your PATH environment variable.)

2. Click **Next**.

The Name the Environment page appears:

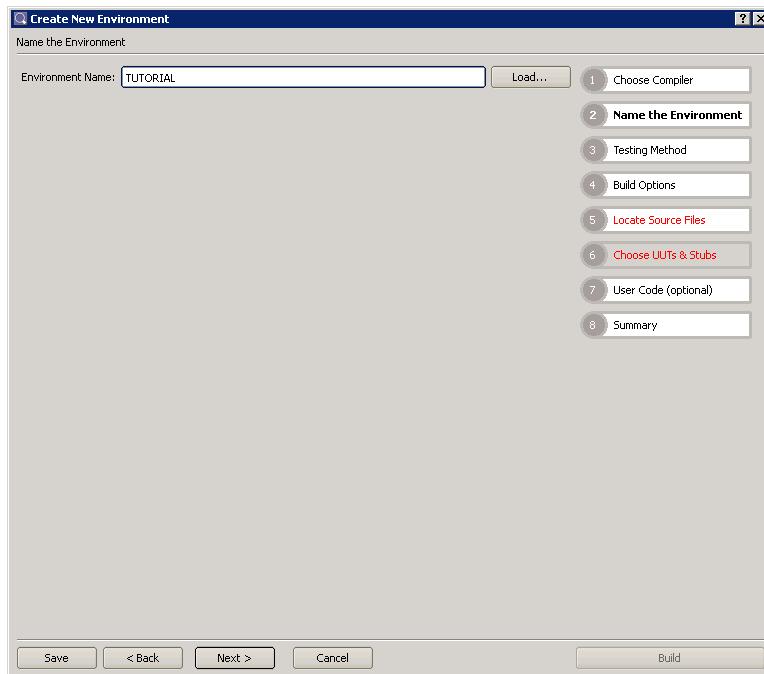


Naming your Test Environment

For the purposes of this tutorial, you will name your environment 'tutorial'.

1. Enter **tutorial** into the Environment Name field.

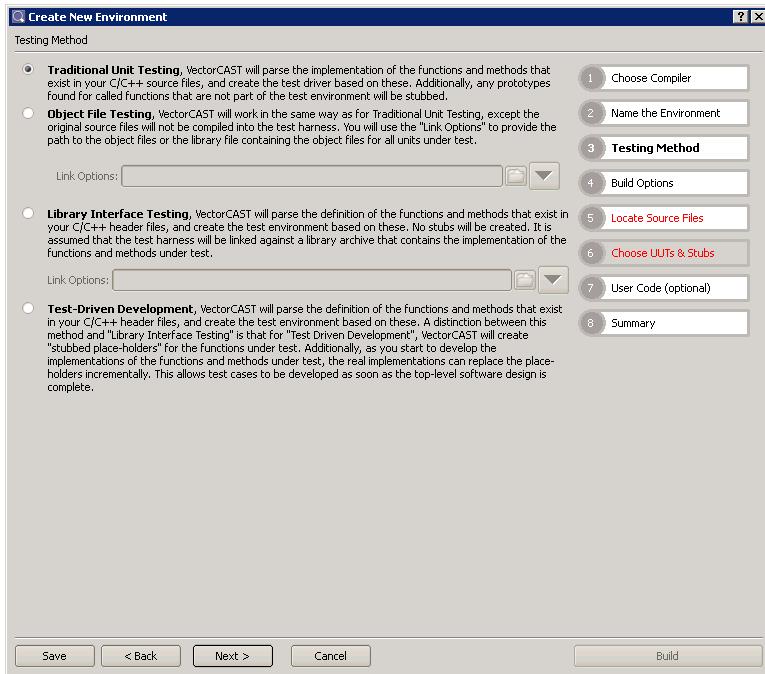
VectorCAST echoes the letters in uppercase:



2. Click **Next**.

Choosing the Testing Method

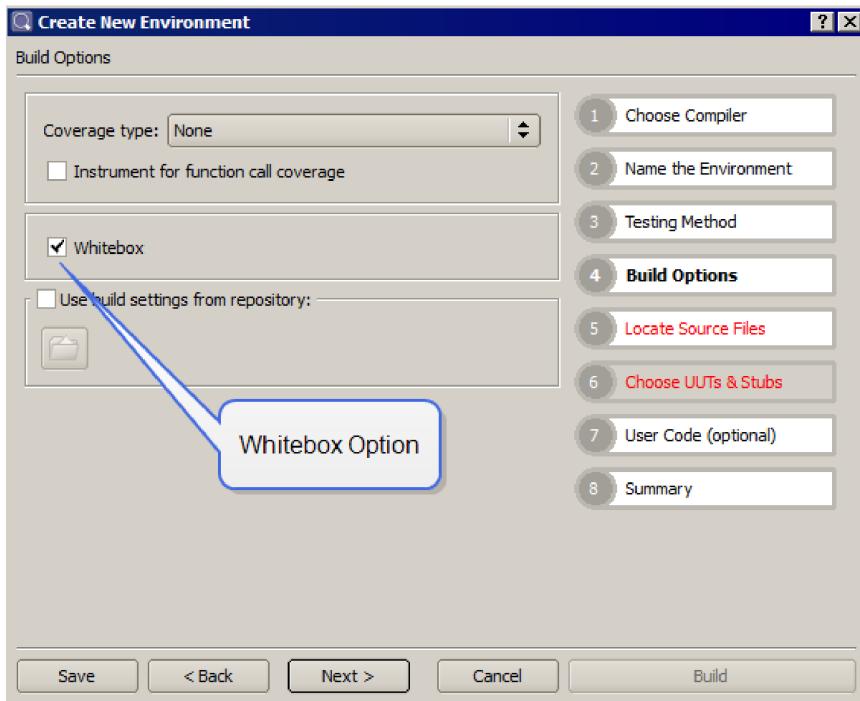
The Testing Method page opens:



This page of the Wizard offers four choices for the method of testing to use. Depending on the testing method, VectorCAST builds a different test harness. For the purposes of this tutorial, we will go with the **Traditional Unit Testing** method.

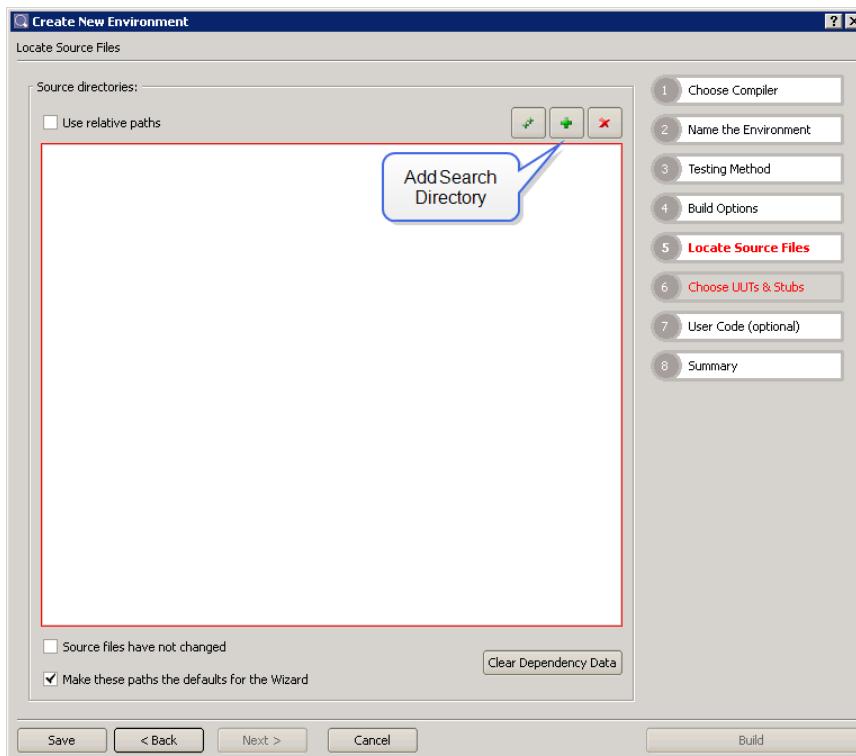
3. Click Next.

The Build Options page opens. Click the checkbox next to “**Whitebox**.” Turning on whitebox ensures you access to private and protected member variables in the source code units.



4. Click **Next**.

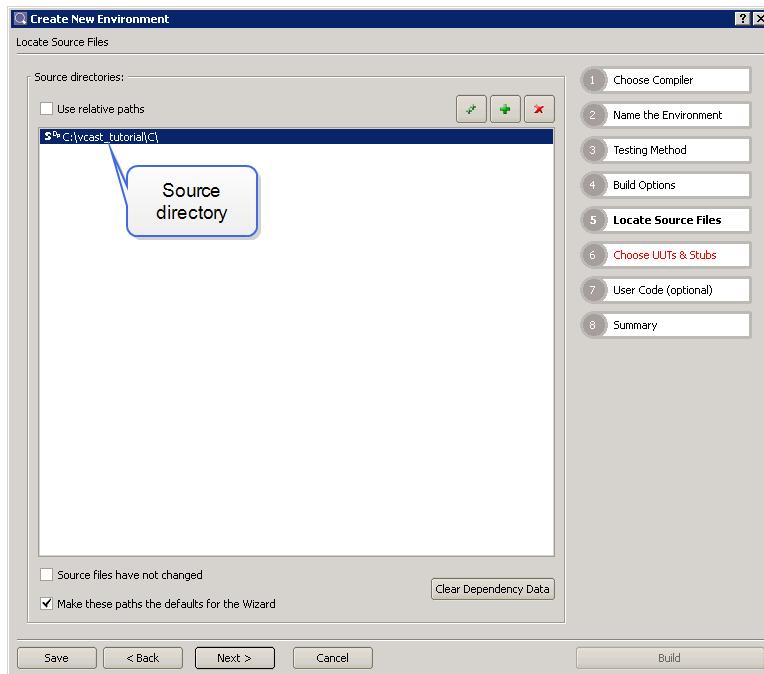
The Locate Source Files page opens:



Specifying the Source Files to be Tested

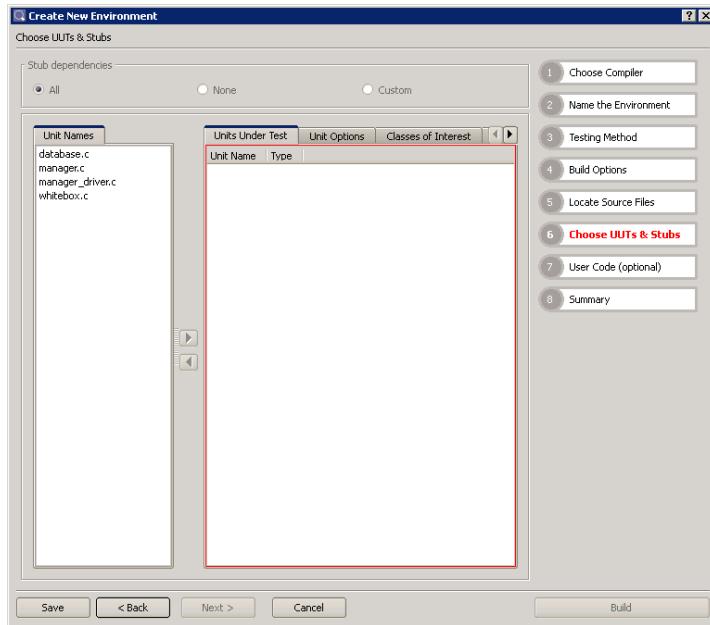
To specify the directory containing the source files to be tested.

1. Click the **Add Search Directory** button  to display the Add Search Directory dialog. The name of the working directory (`C:/vcast_tutorial/c/`) displays in the Look in field; the content of this directory displays in the pane below. In this tutorial, you will be building your environment in the same directory in which the source files are located. In practice, you might need to navigate to a different directory.
2. Click **Choose**. The Search directories pane on the Locate Source Files page displays the name of the source directory. The icon  at the left side of the search directory path indicates it is a **Search** directory, so VectorCAST will search it for files that it needs as it is parsing the Unit Under Test.



3. Click **Next**.

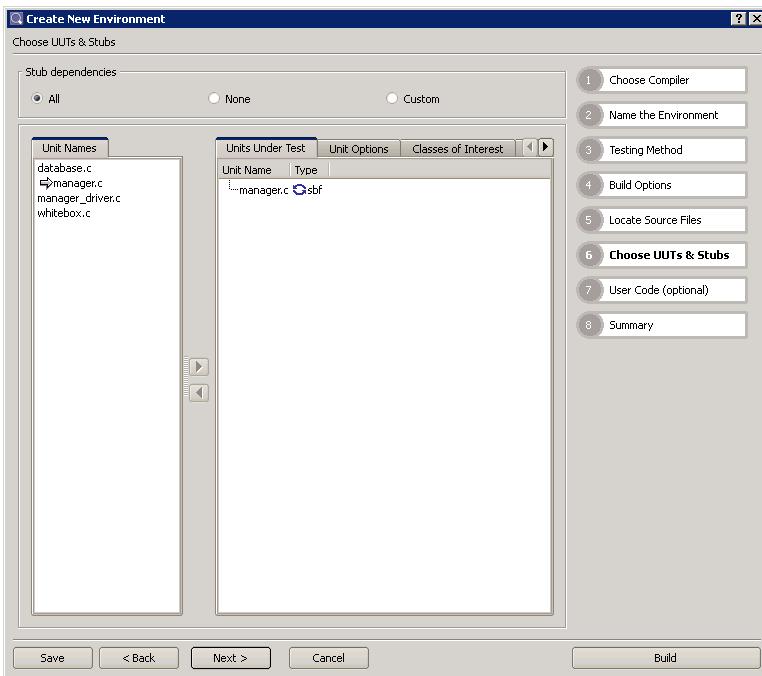
The Choose UUTs & Stubs page appears:



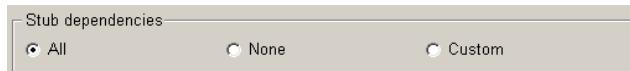
Designating UUTs and Stubs

You use the Choose UUTs & Stubs page to designate units as a UUT or stub and the method of stubbing.

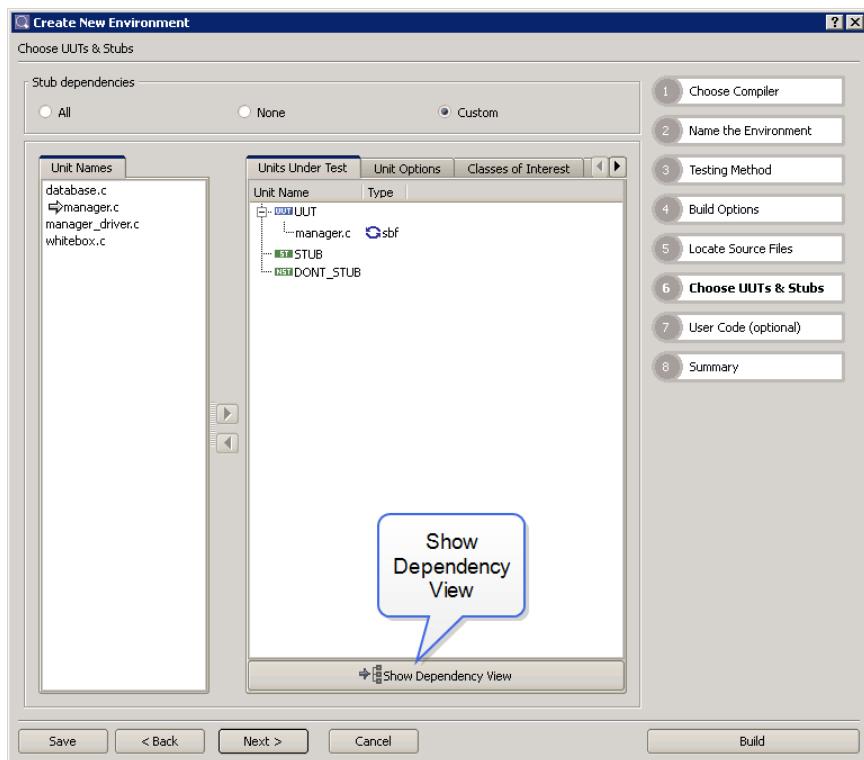
The pane under Unit Names lists the units that VectorCAST found in the specified source directory. You can select one or more of these units to be UUTs. For this tutorial, you will select only manager.



1. Click manager.c under Unit Names, and then click the move-right button; or simply double-click manager.c. The name manager.c moves to the Units Under Test tab, and is displayed with the SBF icon (sbf) next to it. By default, the unit is a UUT with Stub-by-function enabled.
- Three options are available under Stub dependencies for designating stubbing: All, None, or Custom:



As indicated, VectorCAST defaults to stubbing all dependent units. You can override this setting by selecting None or Custom. The Custom option allows you to select individual units for stubbing by implementation or not stubbing.

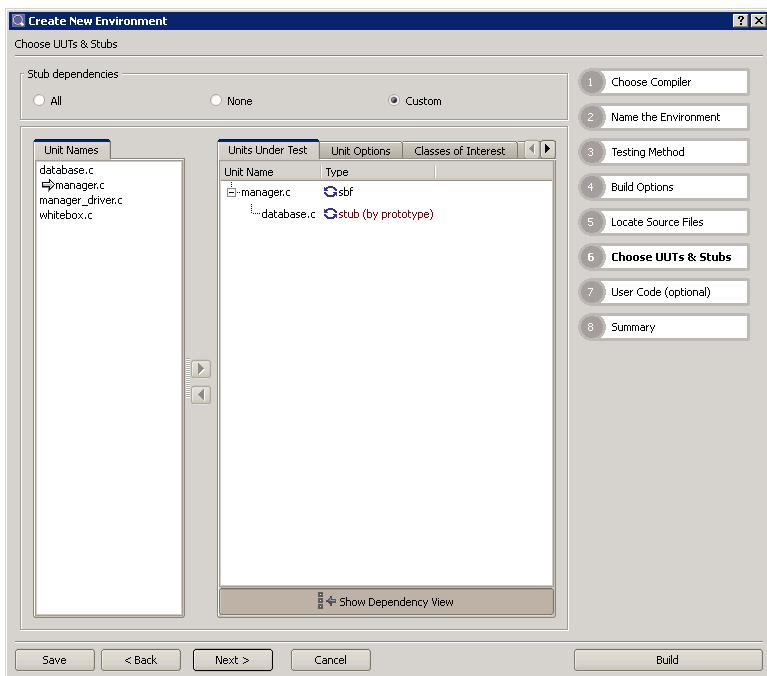
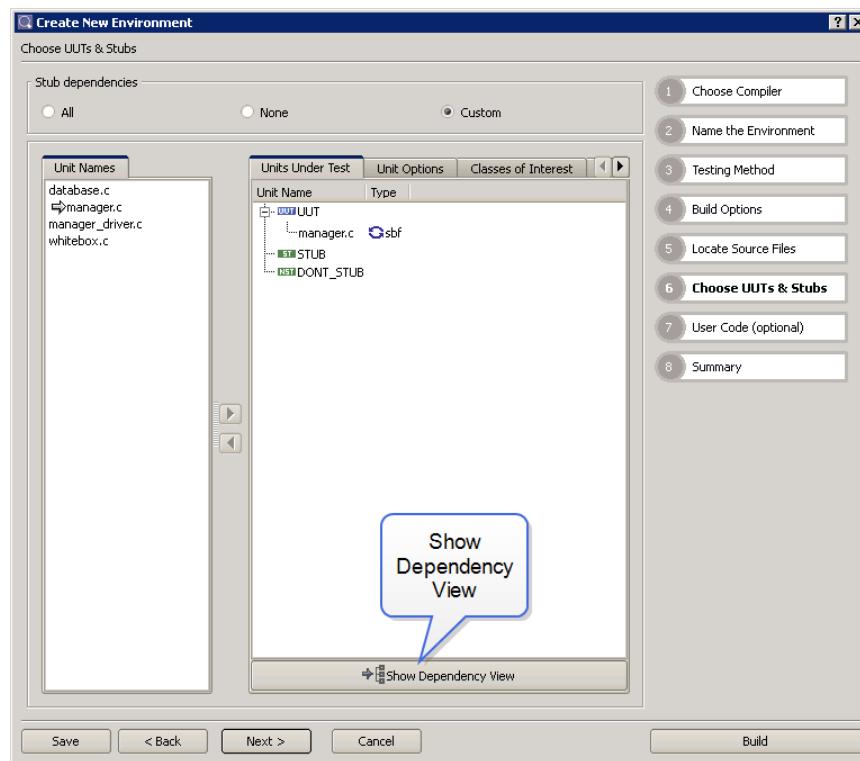


Note: By default, VectorCAST performs stubbing By Prototype.

Steps 2 – 4 are optional.

2. Click the **Custom** radio button.

A toggle button (Show Dependency View ) appears at the bottom of the Units Under Test tab:



3. Click the **Show Dependency View** button.

VectorCAST parses the units in the search directory, and finds that the unit database is a dependent of the unit manager. It draws the hierarchy to represent this relationship between the units.

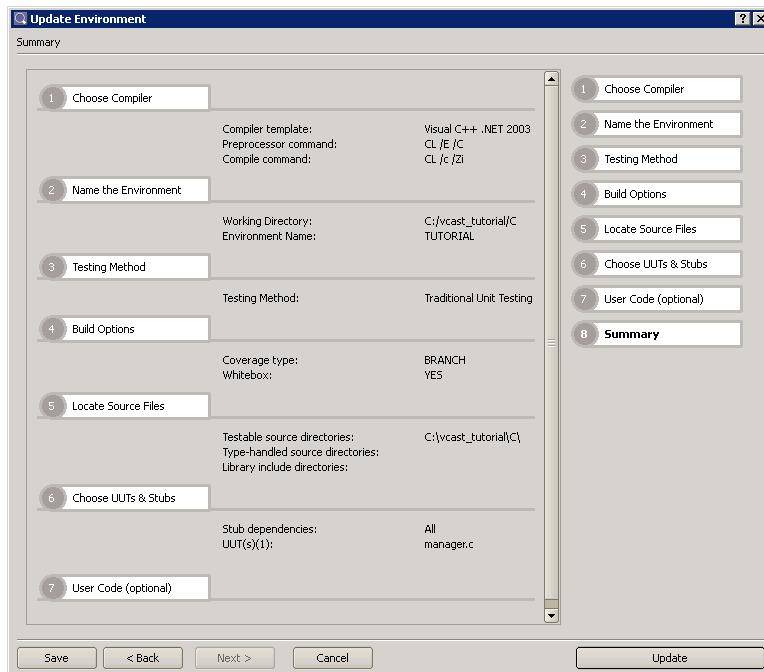
Database appears as a dependent to manager, and “stub (by prototype)” appears in the Type column for database, to indicate that it is going to be stubbed using its prototype:

4. Click the cycle icon  for the unit database.
Rotate through the three options: stub (by implementation), not stubbed, stub (by prototype). Select the default option: stub (by prototype).
5. Click **Next**.
6. The User Code page appears. The User Code page is optional and not needed for this tutorial.

Viewing a Summary of Your Test-Environment Specifications

1. Click **Next**.

The Summary page appears:



This page allows you to check your test-environment specifications before you generate the test harness.

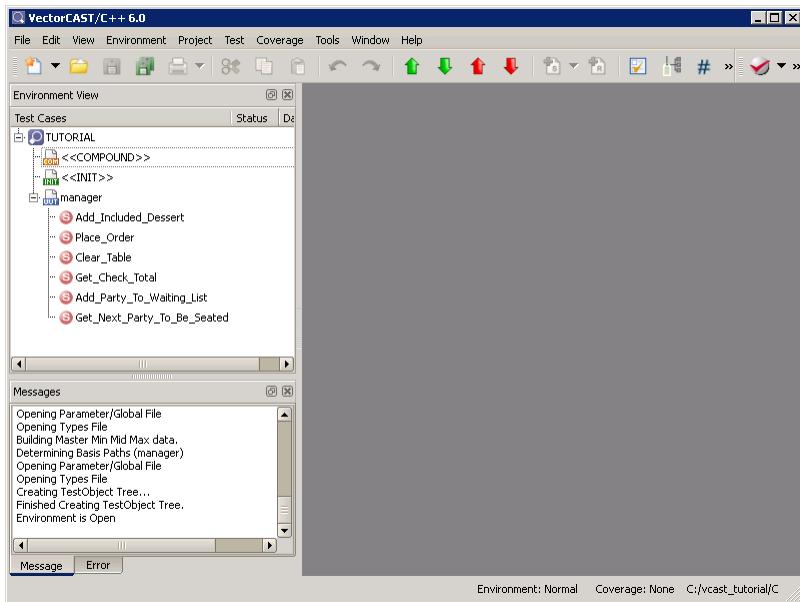
Generating an Executable Test Harness

To compile your specifications into an executable test harness:

1. Click the **Build** button.

As an environment is built, VectorCAST displays a series of messages in the Message window. VectorCAST is parsing **manager.c**, creating a stub for **database.c**, and compiling and linking the test harness.

When processing has completed, VectorCAST displays the main window:



VectorCAST has created an environment file named **TUTORIAL.vce** and a subdirectory named **TUTORIAL**.

The Environment View shows that the current environment (TUTORIAL) has one UUT, named **manager**, which has six subprograms: **Add_Included_Dessert**, **Place_Order**, **Clear_Table**, **Get_Check_Total**, **Add_Party_To_Waiting_List**, and **Get_Next_Party_To_Be_Seated**

At this point, VectorCAST has built the complete test harness necessary to test the **manager** unit, including:

- > A test directory with all test-harness components compiled and linked.
- > The test driver source code that can invoke all subprograms in the file (**manager**).
- > The stub source code for the dependent unit (**database**).

Note what was accomplished during environment creation:

- > The unit under test was parsed. The dependent unit did not have to be parsed, because its prototype was used to determine the parameter profile.
- > The data-driven driver and stub source code were generated.
- > The test code generated by VectorCAST was compiled and linked with the UUT to form an executable test harness.

The harness that VectorCAST builds is *data driven*. This means that data sets can be built to stimulate the different logic paths of the unit under test without ever having to re-compile the harness. You are now ready to build test cases for the subprograms in the unit under test (**manager**).

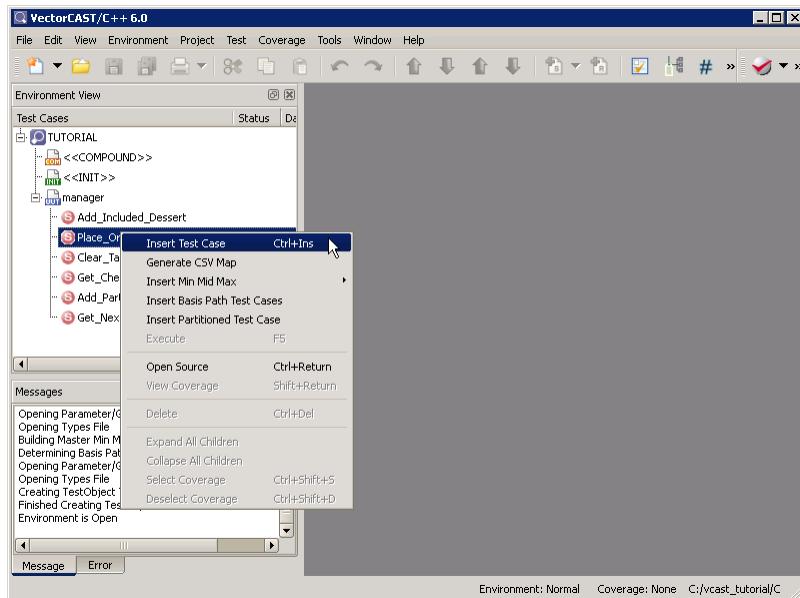
Building Test Cases

In this section, you will build a test case to verify the operation of the subprogram `Place_Order`.

The process of building test cases with VectorCAST is very simple. All the information is available by means of a point-and-click interface. The idea is to create a set of data that will stimulate the unit under test and cause it to execute a particular logic path. In its simplest form, a test case consists of values for the formal parameters of a single subprogram within the unit under test. A more complex test case can also contain values to be returned by a stubbed dependent, or values to be set into global data objects.

Test cases are constructed to stimulate individual subprograms. Each subprogram within a unit will have its own series of test cases defined. The first step is to select a subprogram in the Environment View and create a test case for it.

1. If a list of subprograms is not displayed under `manager` in the Environment View, click the  symbol preceding its name to display the list of subprograms.
2. Right-click `Place_Order` and select **Insert Test Case** from the popup menu:



In the Environment View, `PLACE_ORDER.001` appears as a test case under `Place_Order`.

<code>USER_GLOBALS_VCAST</code>	A collection of global objects for use as a workspace
<code>manager</code>	The unit under test
<code><<SBF>></code>	A collapsed tree of subprograms (other than the subprogram under test) which are available to be stubbed in this test case
<code><<GLOBAL>></code>	A collapsed tree of variables of global scope to the unit under test

Place_Order	The subprogram to be tested
Table	Subprogram parameter of type unsigned short
Seat	Subprogram parameter of type unsigned short
Order	Subprogram parameter of type struct
return	Subprogram parameter of type int
Stubbed Subprograms	A collapsed tree of stubbed subprograms from dependent unit(s)

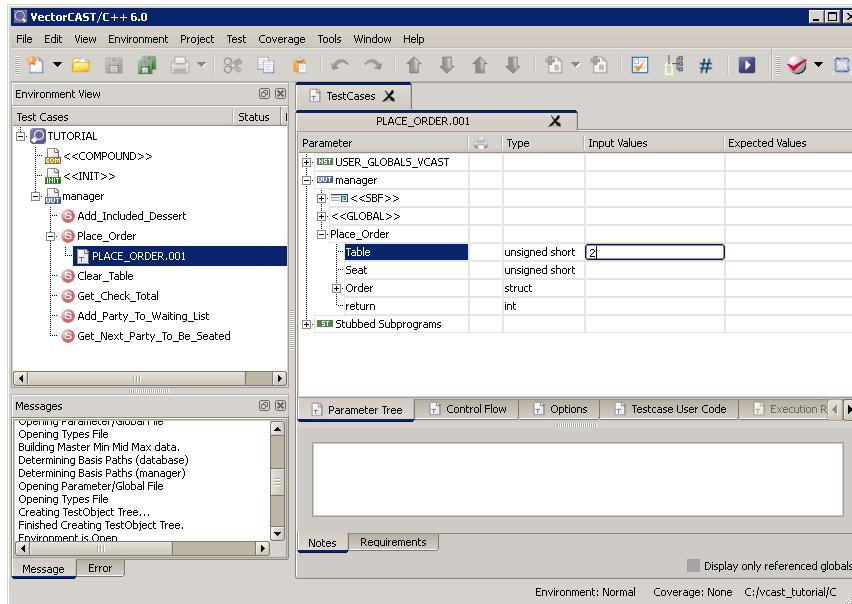


Note: You can change the default name of a test case by right-clicking its name and selecting **Rename** from the popup menu.

The following test case parameter tree appears in the Test Case Editor:

A tree item having branches is preceded by a plus symbol \oplus . To expand a tree item, simply click the \oplus symbol.

- For the parameter **Table**, enter **2** into **Input Values** field:

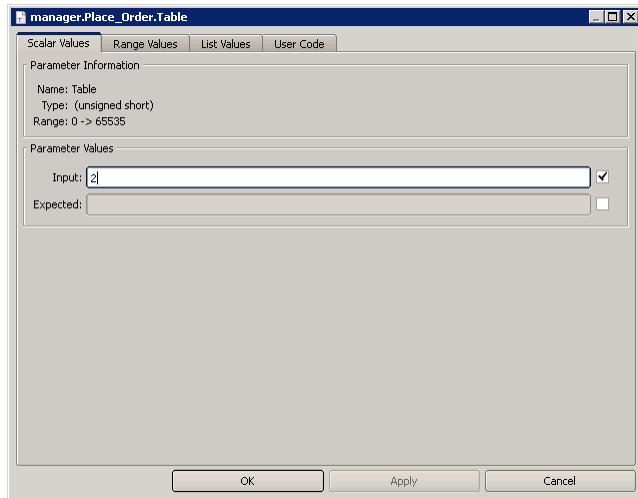


- Press **Enter**.

Table 2 is now the focus of your test case.

- Double-click **Table**.

The parameter dialog box appears:



Notice that you can use this dialog to set simple scalar values, and also to build test cases for a range of values or a list of values, or with user code.

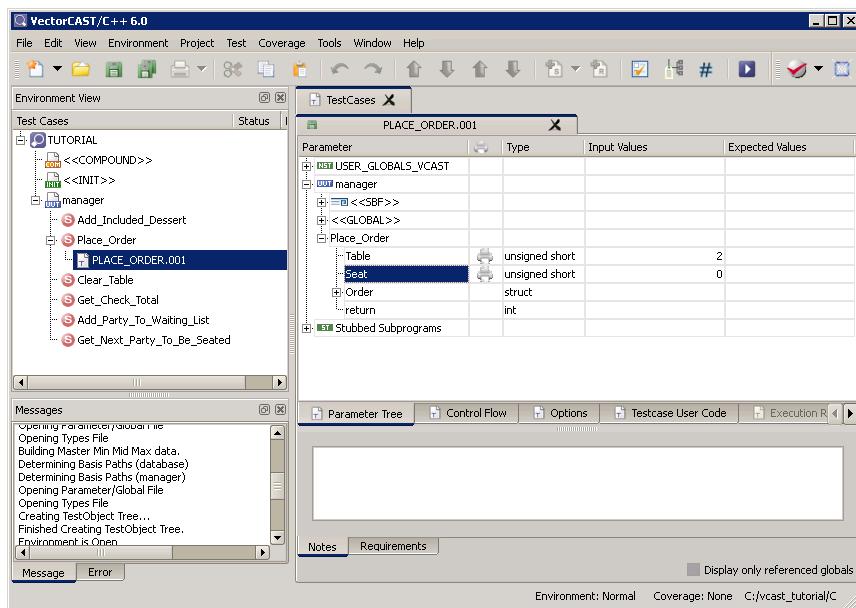
6. Click **Cancel** to dismiss this dialog.

You will use it in the next tutorial.



Note: When you create a test case, the only subprogram displayed for the UUT is the subprogram that this test case has been created for, in this case, Place_Order.

7. For this tutorial, assume now that someone is occupying seat 0 at table 2. Enter **0** into the **Input Values** field for **Seat**, and then press **Enter**.



You will use this same technique to set values for all parameters. Although you can access all parameters of visible subprograms of the unit under test, and of any stubbed unit, it is only necessary to select those parameters that are relevant to the test you are running.

Non-scalar parameters (arrays and structures) are accessed using the same tree-expansion technique that is used for units and subprograms. Parameter **Order** is a structure with five fields. Each field is an enumeration type.

At this point, you could create separate test cases for the other seats at table 2 (as well as for the other tables in the restaurant); however, for the purposes of this basic tutorial, one test case (one order at one seat) is sufficient.

You have someone sitting in seat 0 at table 2; it's time now to take an order.

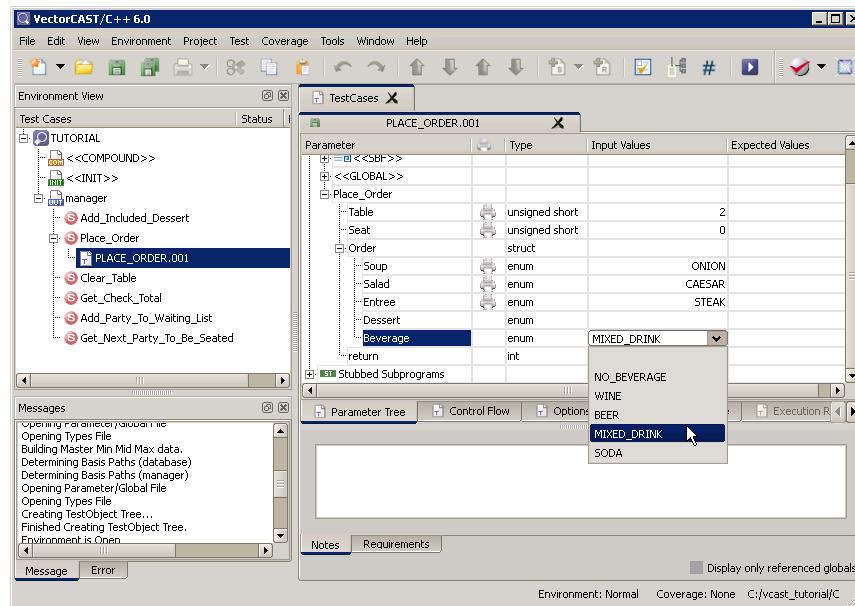
8. Expand **Order** into its fields by clicking .

Five order fields appear: Soup, Salad, Entree, Dessert, and Beverage. (The specifications for these order fields are included in "Tutorial Source Code Listings" on page 311.)

9. For each order field, click in the **Input Values** column and select a value from the drop-down menu that appears.

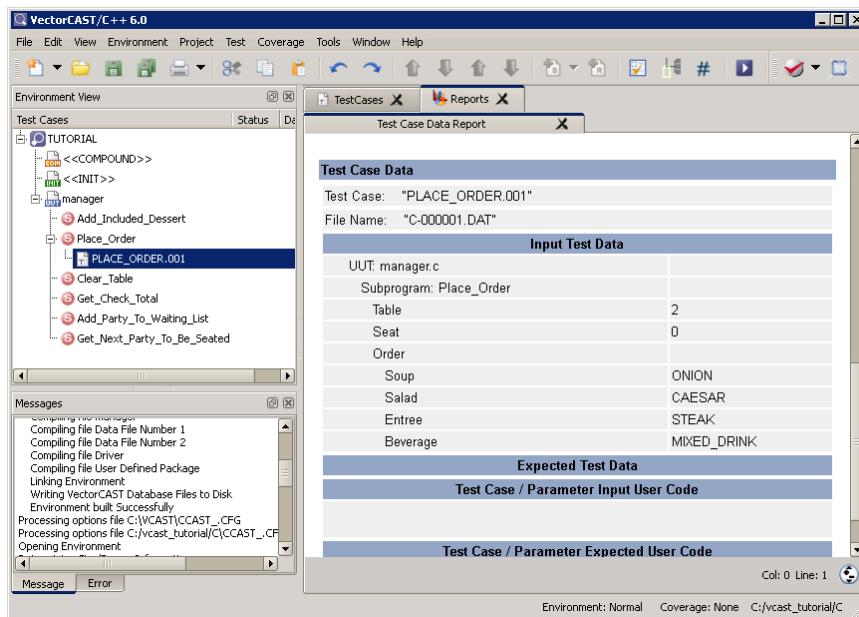
For the purposes of this example, select the following values:

Soup	ONION
Salad	CAESAR
Entree	STEAK
Dessert	(leave unspecified)
Beverage	MIXED_DRINK



 **Note:** To clear a field, select the blank item at the top of the associated drop-down menu.

10. Save test case PLACE_ORDER.001: Either select **File => Save** from the main-menu bar or click the Save  button on the toolbar.
11. To access a test-case data listing, select **Test => View => Test Case Data**. The Test Case Data Report for your test case appears in the Report Viewer.
12. In the Table of Contents section of this report, click **Test Case Data**. The report jumps to the Test Case Data section.



13. Close the report by clicking the X on the report's tab.
- You have now built a test case that is ready to be executed by VectorCAST, without any compiling of data or writing of test code. The data that was set up for this case corresponds to the formal parameter list for subprogram **Place_Order**. You set data for scalar parameters **Table** and **Seat** and for structure parameter **Order**.

At this point you could execute your test case as is, or you could add expected results for comparing against actual results. Because the latter course is generally the more valuable, you will now add expected results to your test case.

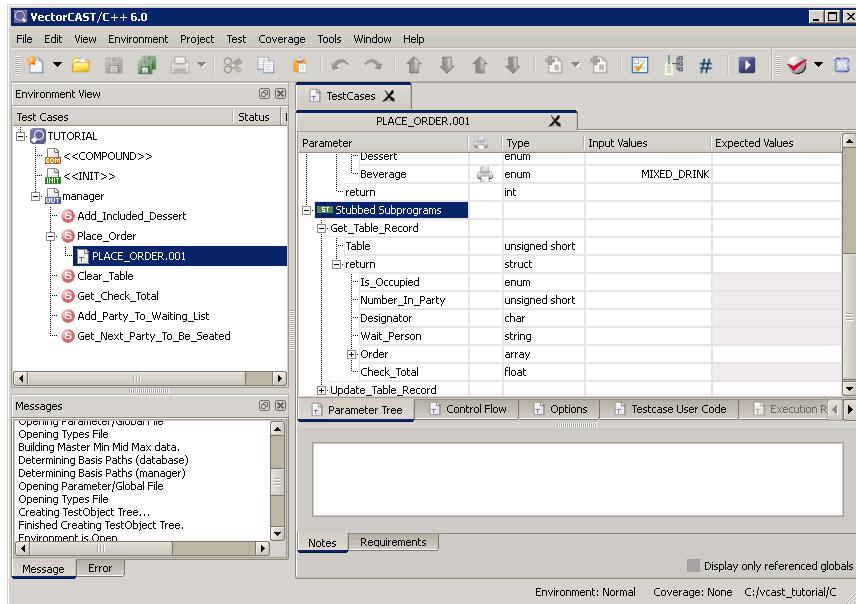
Controlling Values Returned from Stubs

VectorCAST provides you with a simple way to define expected results for each test case. You can add expected results at the time you set input values or at a later time. As a test case is executed, the expected results are compared against actual results to determine whether the test case passed or failed.

In this section, you will add expected results to the test case you created in the previous section:

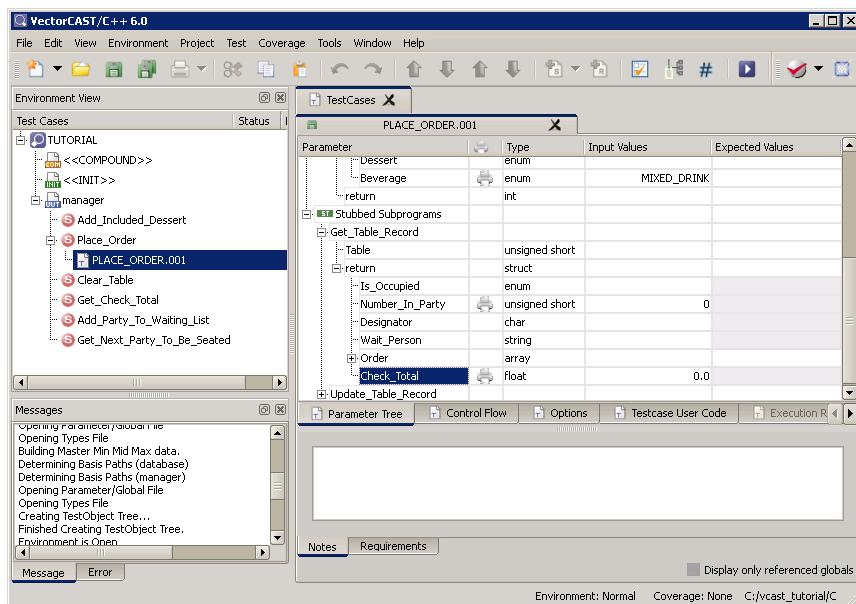
1. Click the  preceding **Stubbed Subprograms**; then click the  preceding **Get_Table_Record**. Then click the  preceding **return**.

The **return** parameter expands into its fields:



Note: If you encounter any problems, refer to the section titled *Troubleshooting*.

2. To initialize the data returned by **Get_Table_Record**, enter **0** into the Input Values fields for **Number_In_Party** and **Check_Total**:



3. Press **Enter**.

The 0 value you entered for `check_Total` becomes 0.0, because `check_Total` is a floating-point number.

You are now ready to enter expected values for comparing against the actual values returned by `Update_Table_Record`.



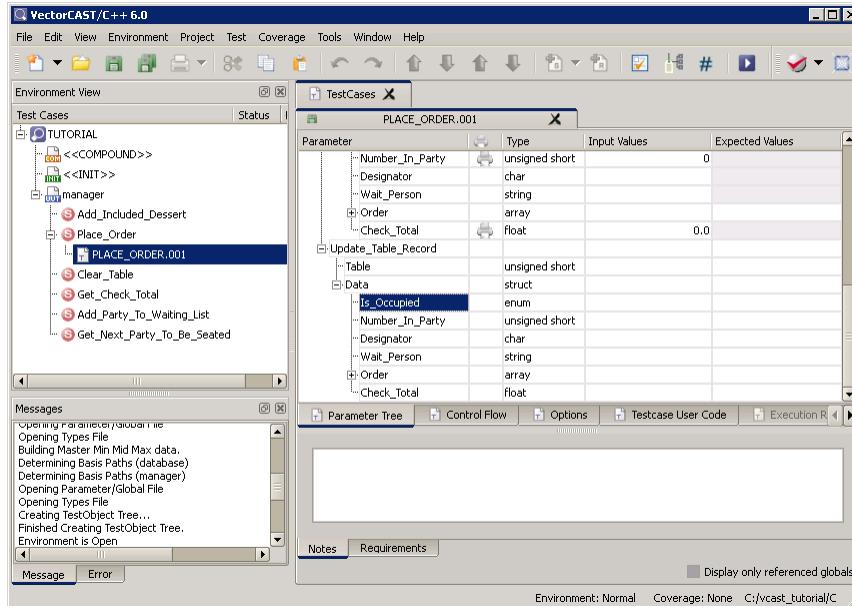
Tip: You might want to make more room by closing up the Parameter trees for both `Get_Table_Record` and `manager`.

Adding Expected Results to the Test Case

VectorCAST provides you with a simple way to define expected results for each test case. You can add expected results at the time you set input values or at a later time. As a test case is executed, the expected results are compared against actual results to determine whether the test case passed or failed.

In this section, you will add expected results to the test case you created in the previous section:

1. Click preceding `Update_Table_Record`, then click preceding `Data`.



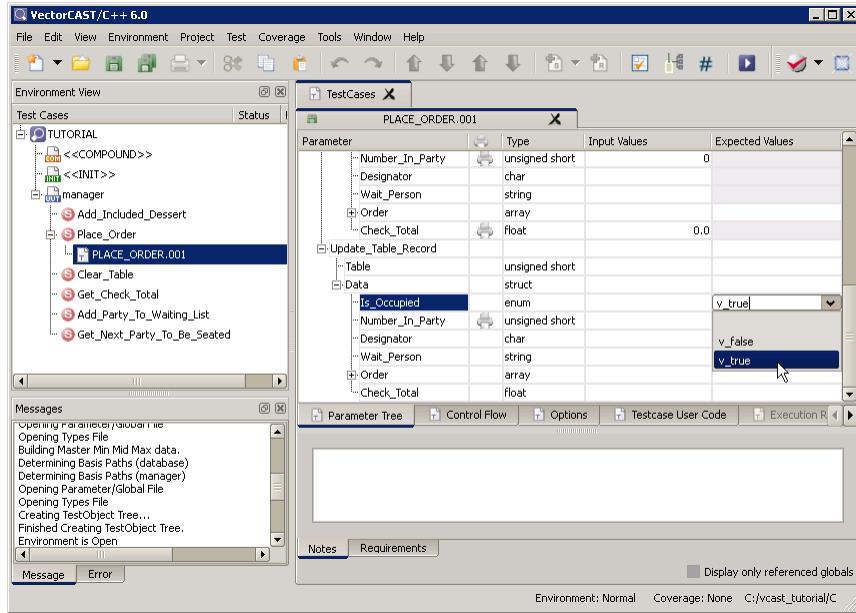
In the **Expected Values** column, enter the following expected results:

`IsOccupied`: **v_true**

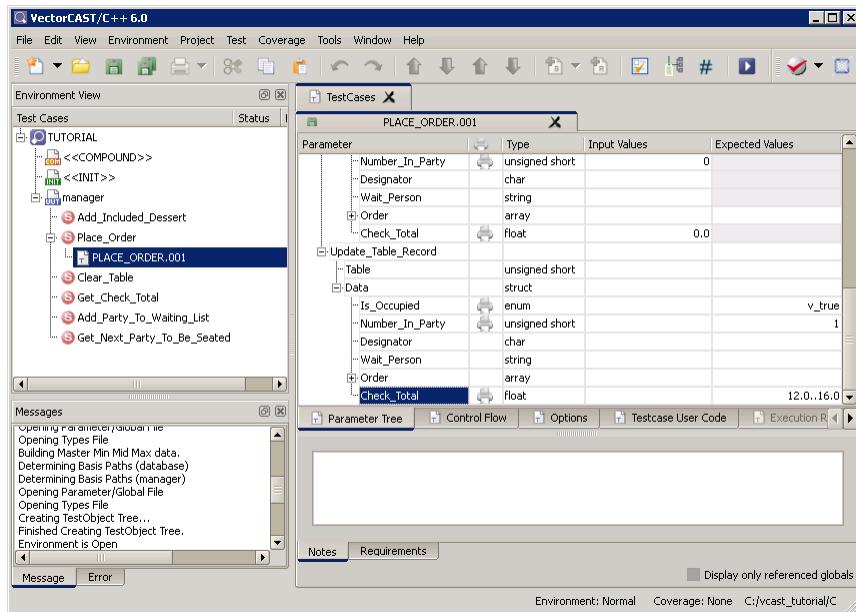
`NumberInParty`: **1**



Note: Be sure you enter these values under **Expected Values** column, not into the **Input Values** column.



2. In addition, enter **12..16** as the range of expected results for **Check_Total**.



3. Expand Order.

Order is an array whose index corresponds to the seat specified in the call to **Place_Order**. Because we specified seat 0, we need to expand **Order[0]**.

4. In the Input Values column, enter **0** next to <>Expand Indices: Size 4>.

Table	unsigned short		
Data	struct		
Is_Occupied	enum	v_true	
Number_In_Party	unsigned short	1	
Designator	char		
Wait_Person	string		
Order	array		
<<Expand Indi...	[0]		
Check_Total	float	12.0..16.0	

5. Press **Enter**.

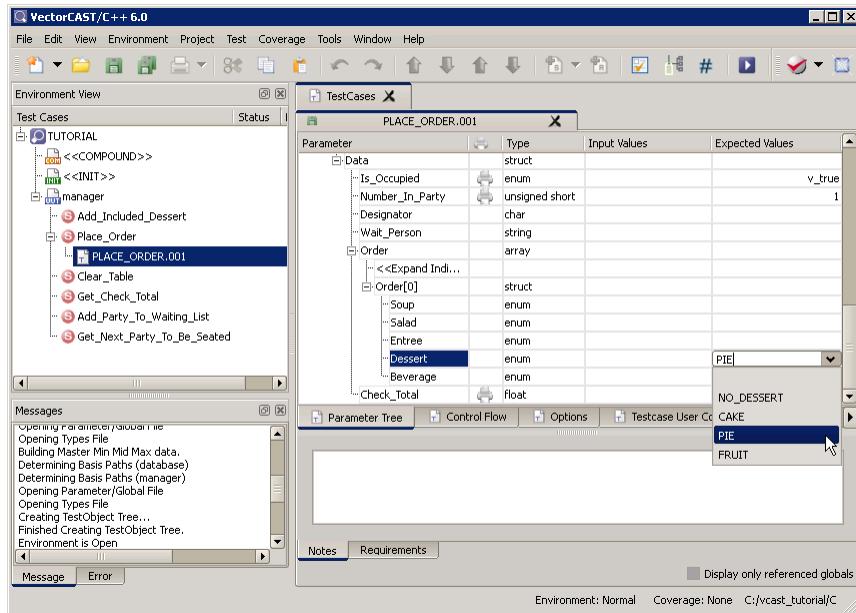
Table	unsigned short		
Data	struct		
Is_Occupied	enum	v_true	
Number_In_Party	unsigned short	1	
Designator	char		
Wait_Person	string		
Order	array		
<<Expand Indi...	[0]		
Order[0]	struct		
Check_Total	float	12.0..16.0	

The array element [0] in the array **Order** is expanded in the Parameter Tree, and the “0” is removed.

6. Expand the tree for **order [0]**.

Wait_Person	string		
Order	array		
<<Expand Indi...			
Order[0]	struct		
Soup	enum		
Salad	enum		
Entree	enum		
Dessert	enum		
Beverage	enum		
Check_Total	float	12.0..16.0	

7. For the Expected Value for Dessert, enter **PIE**. We expect Pie for dessert because **Place_Order** calls **Add_Included_Dessert**, which sets the Dessert to Pie.



You now have a test case defined with input data and expected results.

The values entered into the Expected Values column for a stub are verified as they are passed into the stub; the values entered into the Input Values column are passed out of the stub and back into the test harness.

You must now save your data in order to make them a part of your test case.

When a test case has been modified and needs to be saved, a green icon () appears to the left of its name on a tab above the Test Case Editor

8. To save your modifications, click the **Save** button  on the toolbar.
9. To view the saved data, select **Test => View => Test Case Data**, then click the link **Test Case Data**.

Table 1. Test Case Data

Test Case: "PLACE_ORDER.001"

File Name: "C-000001.DAT"

Input Test Data	
UUT: manager.c	
Subprogram: Place_Order	
Table	2
Seat	0
Order	
Soup	ONION
Salad	CAESAR
Entree	STEAK
Beverage	MIXED_DRINK
Stubbed Subprograms:	

Unit: manager.c	
Subprogram: Get_Table_Record	
return	
Number_In_Party	0
Check_Total	0.0
Expected Test Data	
Stubbed Subprograms:	
Unit: manager.c	
Subprogram: Update_Table_Record	
Data	
Is_Occupied	v_true
Number_In_Party	1
Order	
[0]	
Dessert	PIE
Check_Total	BETWEEN:12.0 AND:16.0
Test Case / Parameter Input User Code	
Test Case / Parameter Expected User Code	



If your test case Data report does not show any data in the section labeled Expected Test Data, refer to the section titled Troubleshooting at the end of this tutorial.

-
- To close this report, click the X on the Test Case Data tab. To close all reports, click on the X on the Reports tab.

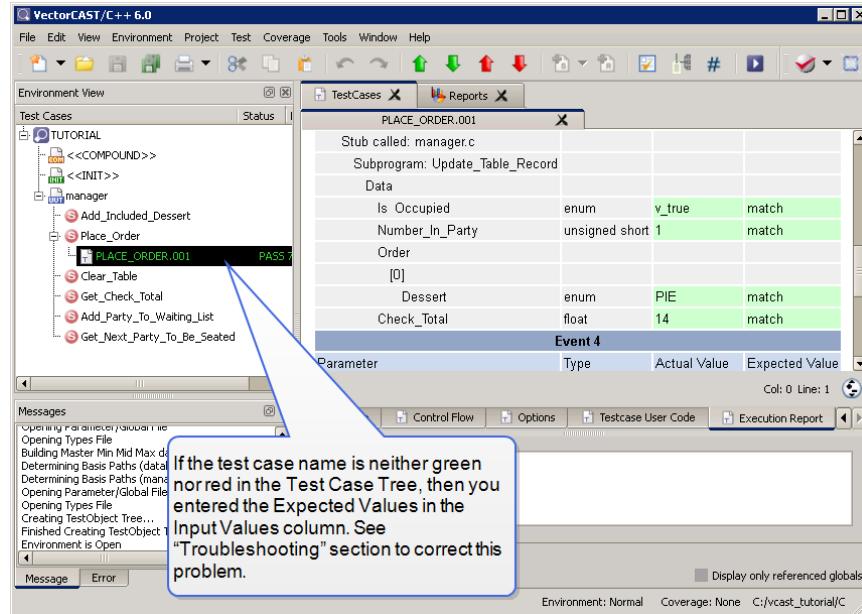
You have now created a test case named PLACE_ORDER.001 that includes expected results. You are now ready to execute it.

Executing Test Cases

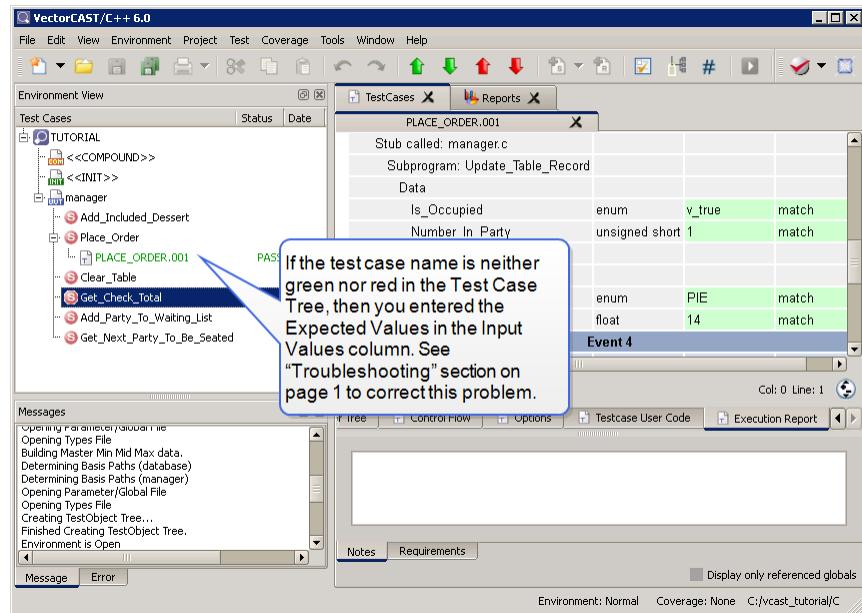
In this section, you will execute the test case you created in the previous sections.

- To execute your test case, click **PLACE_ORDER.001** in the Environment View, then click the **Execute** button
- When processing completes, an execution report appears in the Report Viewer.
- Click the green down-arrow
- or scroll down the report until you come to a group of table cells shaded in green.

The green shading and the term **match** here indicate that the expected values you specified were matched by actual values. Your test passed!



Note that **PLACE_ORDER.001** in the Environment View is now green. This color-marking signifies that the most-recent execution of test case PLACE_ORDER.001 passed. In addition, the Status field displays PASS in green, while the Date field displays the hour-minute-second of the execution.



Generating Test Reports

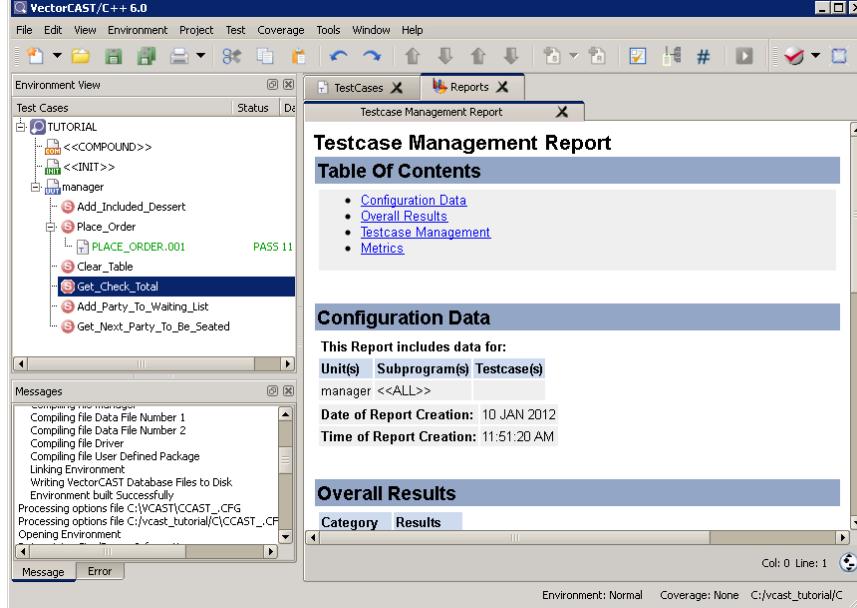
In this section, you will view the Test Case Management Report for the recent execution of test case PLACE_ORDER.001.

A Test Case Management Report includes:

- > The results of all tests conducted with this environment
- > The number of test cases that passed
- > The number of comparisons made that passed
- > The pass-fail status of all test cases
- > Metrics data, including cyclomatic complexity and, if code coverage was enabled, the coverage achieved

1. To view the Test Case Management Report for your environment, select **Test => View => Test Case Management Report** from the main-menu bar.

The Test Case Management Report for environment TUTORIAL appears in the Report Viewer:



Note: You can customize the appearance of this or any other VectorCAST report. To customize a report, select **Tools => Options**; click the **Report** tab; then click the **Format** sub-tab.

2. Use the scrollbar to move through the report.
3. To print your Test Case Management Report, click the **Print** button on the toolbar  , or select **File => Print** from the main-menu bar.
4. To save your report into a file, select **File => Save As**.

When the **Save As** dialog box opens, enter **tutorial_report.html** for the file name, then click **Save**.



Note: You can open a saved report with any HTML browser (Internet Explorer, Mozilla, Netscape, etc.).

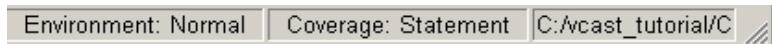
Analyzing Code Coverage

Code coverage analysis enables you to see which parts of your application are being stimulated by your test cases. Once coverage is initialized for an environment, you can choose to run your VectorCAST test cases with either the coverage-instrumented test harness or the uninstrumented harness.

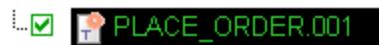
Coverage enables you to view a code coverage report for individual test cases and an aggregate report showing coverage resulting from any subset of test cases.

The Units Under Test must be instrumented for the kind of coverage you want: Statement, Branch, MC/DC, DO-178B Level A, DO-178B Level B, or DO-178B Level C. In this section, you will instrument the source files in your environment for statement coverage:

1. To instrument the source files in your environment for statement coverage, select **Coverage => Initialize => Statement**.
The status bar on the bottom right of the VectorCAST main window now indicates that statement coverage is active:



2. To execute an existing test case and generate a report, click **PLACE_ORDER.001** in the Environment View, then click the execute button ().
When coverage is enabled, VectorCAST keeps track of the lines or branches in the source code that are traversed during execution. You can access a report on the results by way of clicking the checkbox next to the test case name in Environment View.
3. To access the coverage report for your test case, click the **green checkbox** to the left of **PLACE_ORDER.001**.



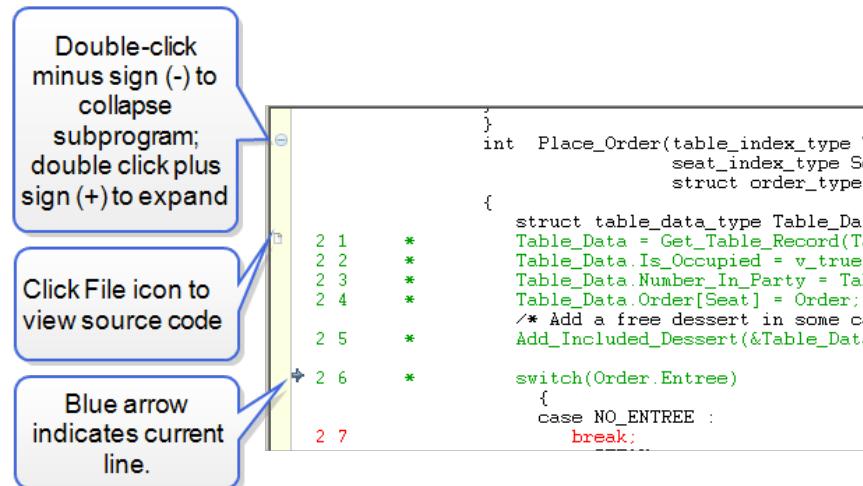
An annotated version of the **manager** source code appears in the Coverage Viewer:

```

VectorCAST/C++ 6.0
File Edit View Environment Project Test Coverage Tools Window Help
Environment View
Test Cases Status
TUTORIAL
  <<COMPOUND>>
    <<INIT>>
      manager
        -> Add_Included_Dessert
        -> Place_Order
          -> PLACE_ORDER_001 PASS 11
            -> Clear_Table
            -> Get_Check_Total
            -> Add_Party_To_Waiting_List
            -> Get_Next_Party_To_Be_Seated
Messages
Compiling VectorCAST...
Compiling File Data File Number 1
Compiling File Data File Number 2
Compiling File Data File Number 3
Compiling file User Defined Package
Linking Environment
Writing VectorCAST Database Files to Disk
Environment built Successfully
Processing options file C:\VCAST\CCAST_.CFG
Processing options file C:/vcast_tutorial/C\CCAST_.CF
Opening Environment
Message Error
Coverage Metrics Basis Paths
Environment: Normal Coverage: Statement C:/vcast_tutorial/C
  
```

Each set of repeating numbers in the first column marks a subprogram within **manager**. The ascending numbers in the second column mark the executable statements within a subprogram. The statements in green have been exercised; the statements in red have not been exercised; the statements in black cannot be exercised, or are the continuation of previous statements.

Note the icons on the left-hand margin of the Coverage Viewer:



A circled minus-sign  means the associated subprogram is fully expanded in the Coverage Viewer, and can be collapsed. To collapse a subprogram (to its first line), double-click either the circle or the first line of the subprogram.

Clicking the file symbol gives you immediate access to the associated source code.

The status bar (upper right) tells you the number and percentage of statements exercised.

4. Hover your mouse over **Statements** on the status bar.

A tool tip provides the exact number of lines covered; in this case, 12 of 40. (30%)



Note: VectorCAST's animation feature allows you to view the step-by-step coverage of a test case. This feature is described in the user guide accompanying your version of VectorCAST.

The Metrics tab at the bottom of the Coverage Viewer displays a tabular summary of the coverage achieved, on a per subprogram basis.

5. Click the **Metrics** tab:

Subprogram Name	Complexity V(a)	Statements
Add_Included_Dessert	3	2 / 4
Place_Order	5	10 / 17
Clear_Table	2	0 / 8
Get_Check_Total	1	0 / 2
Add_Party_To_Waiting_List	3	0 / 6
Get_Next_Party_To_Be_Seated	2	0 / 3
TOTAL	16	12 / 40

This table tells you that subprogram `Place_Order` consists of 17 executable statements; that 10 of these statements were exercised during the selected test; and that the complexity of the subprogram is 5, meaning there are five distinct execution paths.

6. Click the **Coverage** tab to return to the previous view.

For each green line, you can identify the test case that most recently exercised it.

7. Hold your mouse over any green line in the Coverage Viewer.

A tool tip appears showing the name of the test case that exercised this line:

```

2 1  *
2 2  *
2 3  *
2 4  *
2 5  *
2 6  *
{
    struct table_data_type Table_Data;
    Table_Data = Get_Table_Record(Table);
    Table_Data.Is_Occupied = v_true;
    Table_Data.Number_In_Party = Table_Data.Number_
    Table_Data.Order[Seat] = Order;
    /* Add a free dessert in some cases */
    Add_Included_Dessert(&Table_Data.Order[Seat]);
}
switch(Order.Entree)
{

```



Note: If you had run multiple test cases with coverage enabled, and had multiple check boxes selected in the Environment View, the tool tip would list all cases that provided coverage for the selected line.

- To access the test case that exercised a line, right-click on the line, then select the test case from the popup menu that appears; in this example, select **PLACE_ORDER.001**:

```

2 1  *
2 2  *
2 3  *
2 4  *
2 5  *
2 6  *
{
    struct table_data_type Table_Data;
    Table_Data = Get_Table_Record(Table);
    Table_Data.Is_Occupied = v_true;
    Table_Data.Expand_Subprograms = Table_Data.Number_
    Table_Data.Collapse_Subprograms = Order;
    /* Add a free dessert in some cases */
    Add_Included_Dessert(&Table_Data.Order[Seat]);
}
switch(Order.Entree)
{

```

The test case **PLACE_ORDER.001** opens in the Test Case Editor.

- To return to the annotated source code, click the **Coverage** tab again.

You initialized the source code in your environment for coverage and executed the test cases. You can now view the coverage information that was generated.

Generating Coverage Reports

In this section, you will view the Aggregate Coverage Report for your test environment.

An Aggregate Coverage Report includes:

- > A source-listing for each UUT, indicating the lines covered
 - > A metrics table giving the complexity and the level of coverage for each subprogram
- To view the Aggregate Coverage Report for your test environment, select **Test => View => Aggregate Coverage Report** from the main-menu bar.

The Report Viewer displays a report showing the total coverage for all test cases executed on manager. The associated source code displayed is color-coded to mark the exercised lines (green) and the un-exercised lines (red):

The screenshot shows the VectorCAST/C++ 6.0 interface. The left pane displays the Test Cases view with a tree structure of test cases under the TUTORIAL project. A specific test case, PLACE_ORDER_001, is selected and marked as 'PASS'. The right pane shows the 'Aggregate Coverage' report for the 'manager' unit. The report includes the following details:

- Code Coverage for Unit: manager**
- Coverage Type: statement
- Unit: manager
- Test Case: Aggregate

```

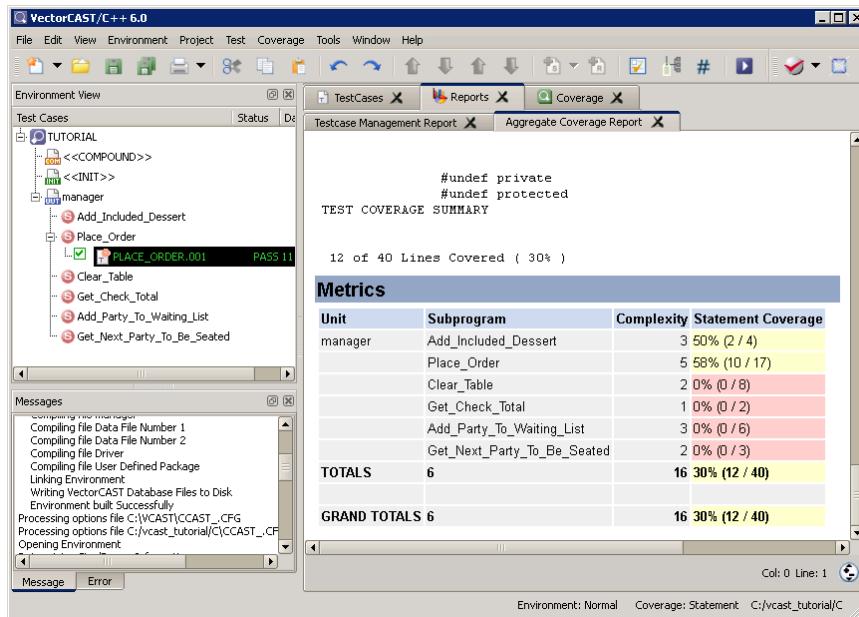
-- Coverage Type: statement
-- Unit: manager
-- Test Case: Aggregate
#define private public
#define protected public
#include "C:\vcast_tutorial\C\ctypes.h"
struct table_data_type Get_Table_Record(table_index_type Table, s
/* Allow 10 Parties to wait */
static name_type WaitingList[10];
static unsigned int WaitinglistSize = 0;
static unsigned int WaitinglistIndex = 0;
const struct order_type NULL_ORDER =
    (NO_SOUP, NO_SALAD, NO_ENTREE, NO_DESSERT, NO_BEVERAGE);
/* This function will add a free dessert to specific entree, salad, and beverage choice */
void Add_Included_Dessert(struct order_type* Order)
{
    if(Order->Entree == STEAK &&
        Order->Salad == CAESAR &&
        Order->Beverage == MIXED_DRINK)

```

The code editor window shows the C code for the 'Aggregate' test case. Colored highlights indicate coverage status: green for exercised lines and red for unexercised lines. The bottom status bar indicates 'Environment: Normal' and 'Coverage: Statement'.

2. Scroll down the report until you come to a table similar to the metrics table you accessed from the Metrics tab.

The partially exercised subprograms, `Add_Included_Dessert` and `Place_Order`, are shown in yellow; the other subprograms, all unexercised, are shown in red. If 100% coverage for a subprogram is achieved, it shows in green:



3. To print the Aggregate Coverage Report, click the **Print** button on the toolbar , or select **File => Print** from the main-menu bar.
 4. To save your report into a file, select **File => Save As**. When the Save As dialog box opens, enter `tutorial_coverage.html` for the file name, then click the **Save** button .
- You can open your saved report with any HTML browser (Internet Explorer, Mozilla, Netscape, etc.).
5. Choose **File => Close All** in preparation for the next section.

Using Stub-By-Function

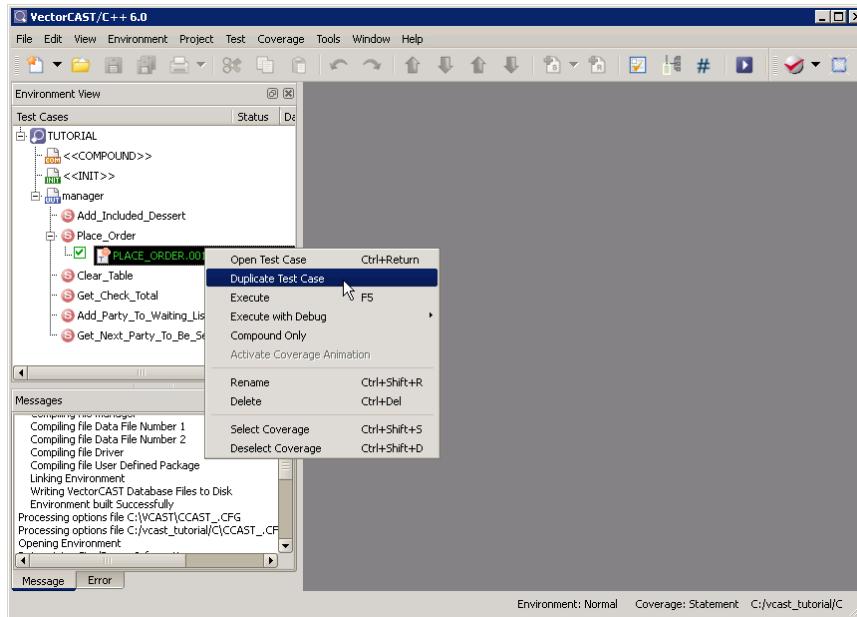
In the last section, we tested the subprogram `Place_Order` using inputs of STEAK, CAESAR, and MIXED_DRINK for the Order. During test execution, `Place_Order` calls the subprogram `Add_Included_Dessert`, which checks for the combination of STEAK and CAESAR salad and MIXED_DRINK. It finds them and sets the Dessert to Pie. However, we would like to test for the case when `Add_Included_Dessert` doesn't set the Dessert to Pie.

In this section, we will create a new test case which stubs the subprogram `Add_Included_Dessert`, so that we can cause it to return something else for the Dessert, such as CAKE.

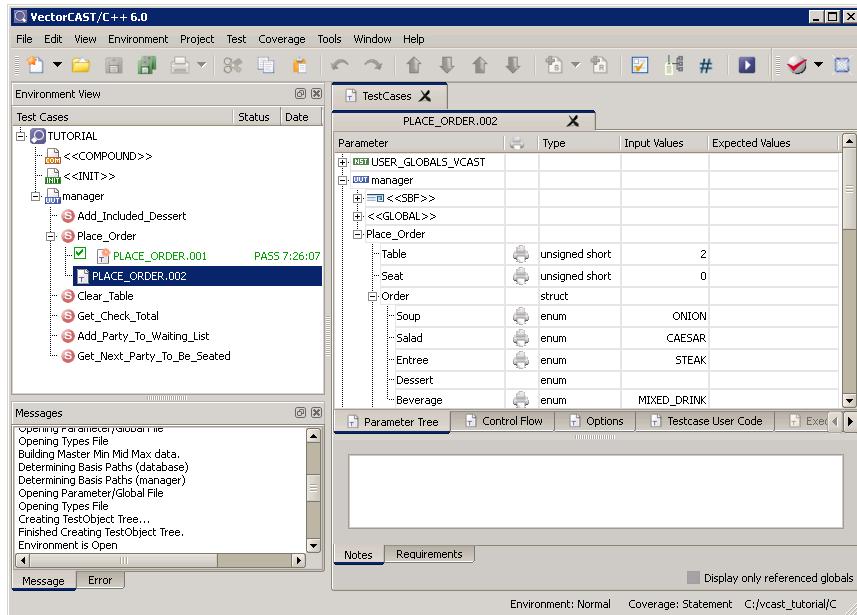
You will use VectorCAST to:

- > Duplicate a test case
- > Stub a subprogram in the UUT

- > Set the value of a parameter in the stubbed subprogram to the desired value, so that it is returned to the test harness instead
 - > Export a test script to see how SBF is specified in a test script
1. In the Environment View, right-click the test case name "PLACE_ORDER.001" and choose **Duplicate Test Case**.



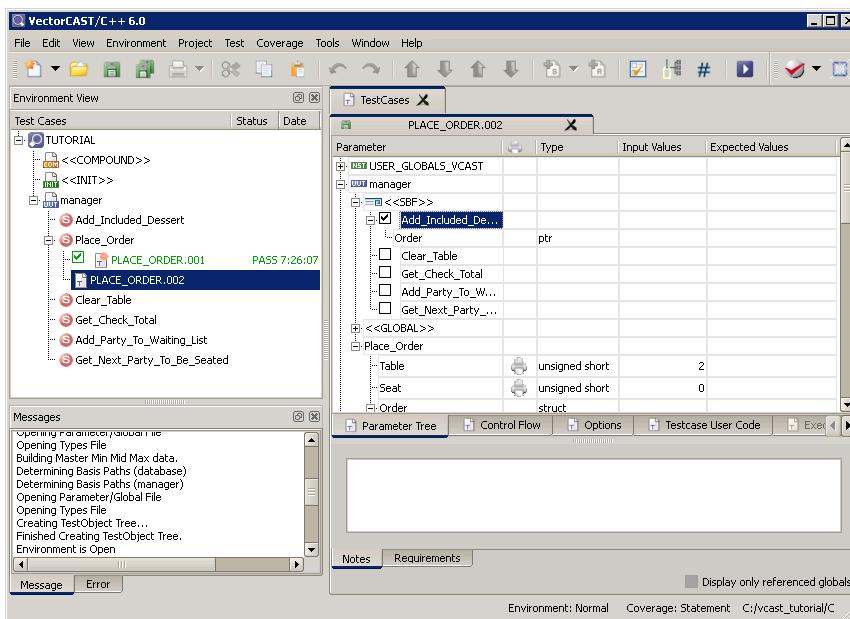
A new test case is created, named PLACE_ORDER.002.





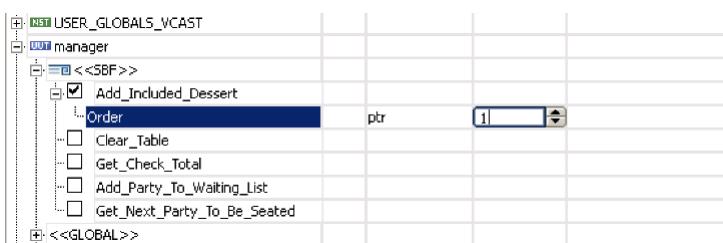
Tip: Using drag-and-drop, you can duplicate a test case in another subprogram. Select the test case, press Alt+Shift, drag-and-drop the test case on the destination subprogram. Only data that applies to the parameters in the new subprogram are copied to the new test case.

2. Click preceding <<SBF>>.
- The list of subprograms defined in the UUT is displayed, not including the subprogram under test, **Place_Order**. The subprograms in this list are available for stubbing in this test case.
3. Click the checkbox next to **Add_Included_Dessert**. The checkmark indicates that this subprogram should be stubbed during test execution.



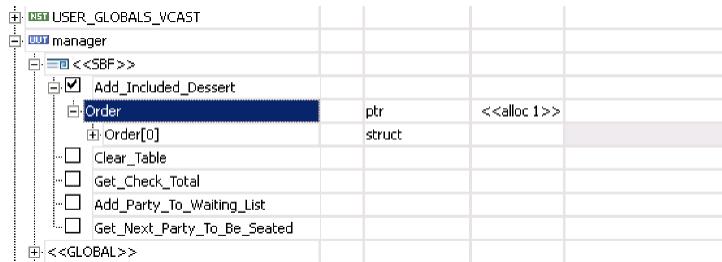
The parameter for **Add_Included_Dessert** is displayed. It is a pointer to the array **Order**. We need to specify a value for **Dessert** for this stubbed subprogram to return to the test harness.

4. To allocate one element of the array, enter 1 in the Input Values column for the pointer to **Order**.



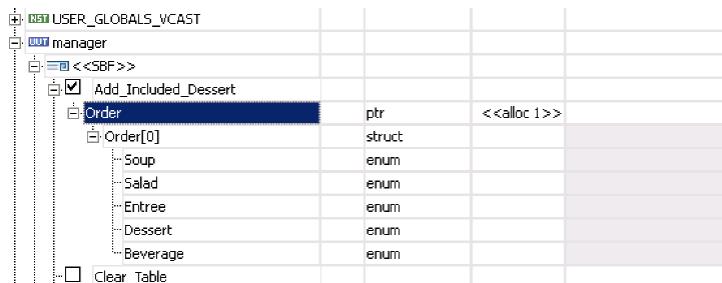
5. Press Enter.

The text in the cell changes to “<<alloc 1>>” and a structure Order[0] is displayed.

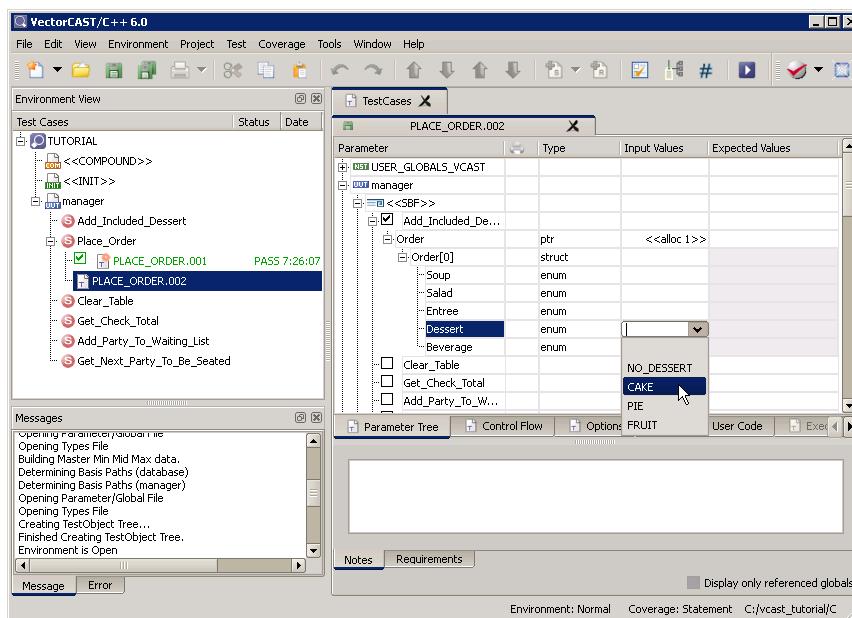


6. Click preceding Order[0].

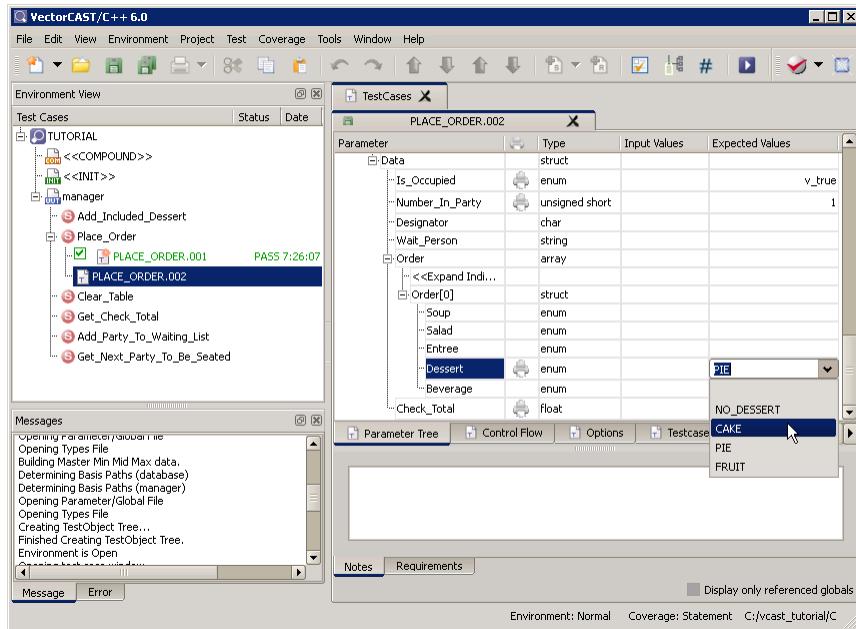
The enumerated elements of the structure are displayed.



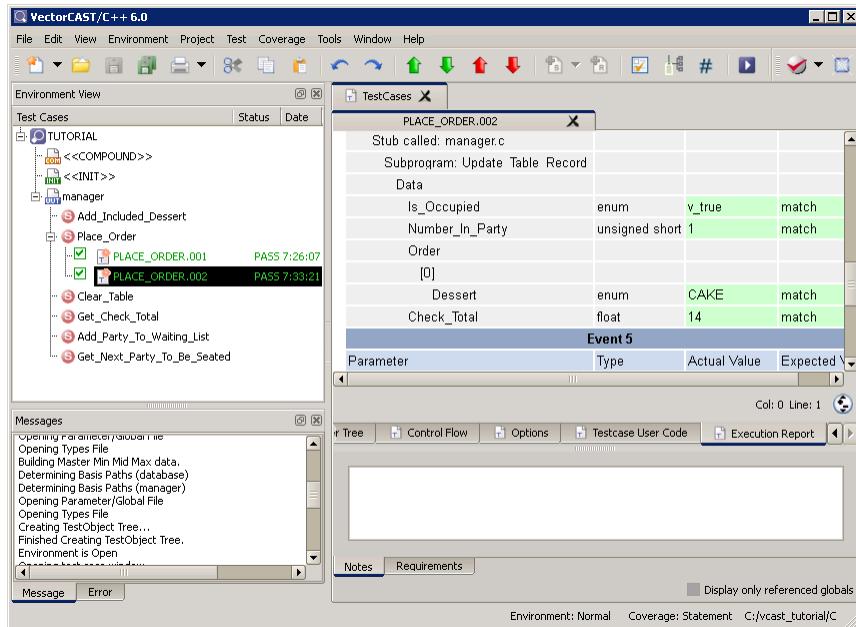
7. Set the Input Value for Dessert to CAKE. When the test executes, CAKE will be returned from **Add_Included_Dessert**, instead of the non-stubbed return value, PIE.



8. Scroll to the end of the Parameter Tree and locate **Update_Table_Record**, which is part of Stubbed Subprograms.
9. Change the Expected Value of the Dessert from **PIE** to **CAKE**.



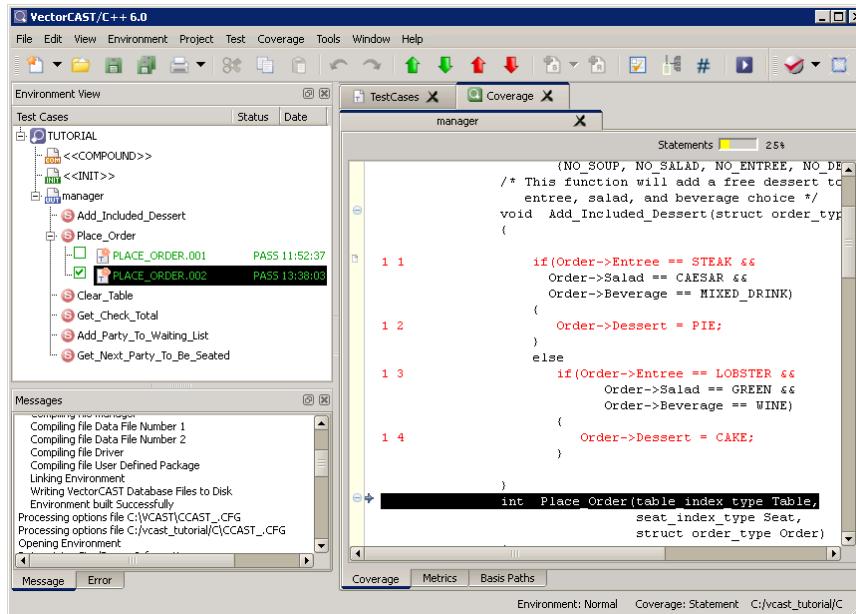
10. To save the test case, click the **Save** button on the Toolbar .
11. Execute the test case by clicking the **Execute** button .
12. The Execution Report tab comes to the top, displaying the results of the test execution. Scroll down to the end of the report to see that the expected value of Cake was matched.



13. To access the coverage achieved for this test case, click the green checkbox to the left of PLACE_ORDER.002.

An annotated version of the manager source code appears in the Coverage Viewer, showing the aggregate coverage achieved by both test cases.

14. Uncheck the green checkbox next to PLACE_ORDER.001.



In this test case, the subprogram Add_Included_Dessert is not covered, as evidenced by statements 1.1 and 1.2 in red. The subprogram does not have coverage because it was stubbed.

during the execution of PLACE_ORDER.002.

15. In the Environment View, select the test case **PLACE_ORDER.002**, then choose **Test => Scripting => Export Script...**.
The Select Script Filename dialog appears.
16. Give the test script a name, such as **PLACE_ORDER.002.tst**, and click **Save**.
17. To open the test script, choose **File => Open**, set the File type to “Script Files (*.env *.tst)” and then select **PLACE_ORDER.002.tst**. Click **Open**.

```
-- VectorCAST 6.0 (01/05/12)
-- Test Case Script
--
-- Environment      : TUTORIAL
-- Unit(s) Under Test: manager
--
-- Script Features
TEST.SCRIPT_FEATURE:C_DIRECT_ARRAY_INDEXING
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2
TEST.SCRIPT_FEATURE:OVERLOADED_CONST_SUPPORT
TEST.SCRIPT_FEATURE:UNDERSCORE_NULLPTR
--

-- Test Case: PLACE_ORDER.002
TEST.UNIT:manager
TEST.SUBPROGRAM:Place_Order
TEST.NEW
TEST.NAME:PLACE_ORDER.002
TEST.STUB:manager.Add_Included_Dessert
TEST.VALUE:manager.Place_Order.Table:2
TEST.VALUE:manager.Place_Order.Seat:0
TEST.VALUE:manager.Place_Order.Order.Soup:ONION
TEST.VALUE:manager.Place_Order.Order.Salad:CAESAR
TEST.VALUE:manager.Place_Order.Order.Entree:STEAK
TEST.VALUE:manager.Place_Order.Order.Dessert:CAKE
TEST.VALUE:manager.Place_Order.Order.Beverage:MIXED_DRINK
TEST.VALUE:uut_prototype_stubs.Get_Table_Record.return.Number_In_Party:0
TEST.VALUE:uut_prototype_stubs.Get_Table_Record.return.Check_Total:0.0
TEST.EXPECTED:uut_prototype_stubs.Update_Table_Record.Data.Is_Occupied:v_true
TEST.EXPECTED:uut_prototype_stubs.Update_Table_Record.Data.Number_In_Party:1
TEST.EXPECTED:uut_prototype_stubs.Update_Table_Record.Data.Order[0].Dessert:CAKE
TEST.EXPECTED:uut_prototype_stubs.Update_Table_Record.Data.Check_Total:12.0..16.0
TEST.END
```

The line beginning with TEST.STUB indicates that **Add_Included_Dessert** should be stubbed during this test's execution. The TEST.VALUE line in bold specifies the Input Value that the stubbed subprogram returns to the test harness during this test's execution. This is possible only because **manager** is a Stubbed-by-Function (SBF) type of UUT, as specified during environment build.

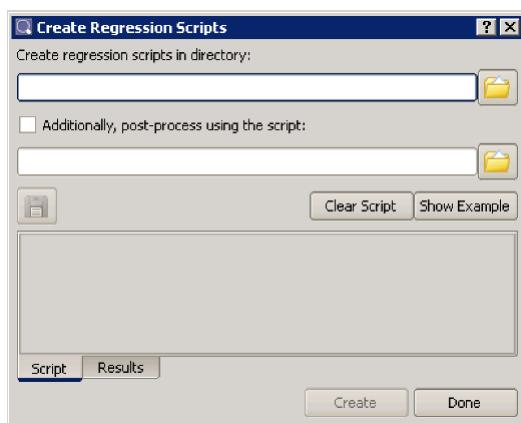
Creating Regression Scripts

Regression Scripts are the files needed to recreate the test environment at a later time. For a typical unit test environment, only 3 files are needed:

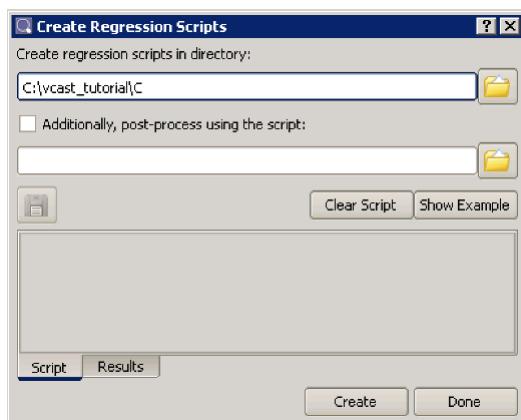
- > A shell script (.bat file on Windows, .sh or .csh on UNIX) sets the options, builds the environment, imports and executes the test cases, and generates reports
- > A test script (.tst) defines test cases to be imported by the shell script.
- > The environment script (.env) defines the settings for the environment, such as the UUT(s), which UUTs are Stub-by-Function, what is the method of stubbing, and where are the search directories.

1. Select **Environment => Create Regression Scripts....**

The Create Regression Scripts dialog appears.

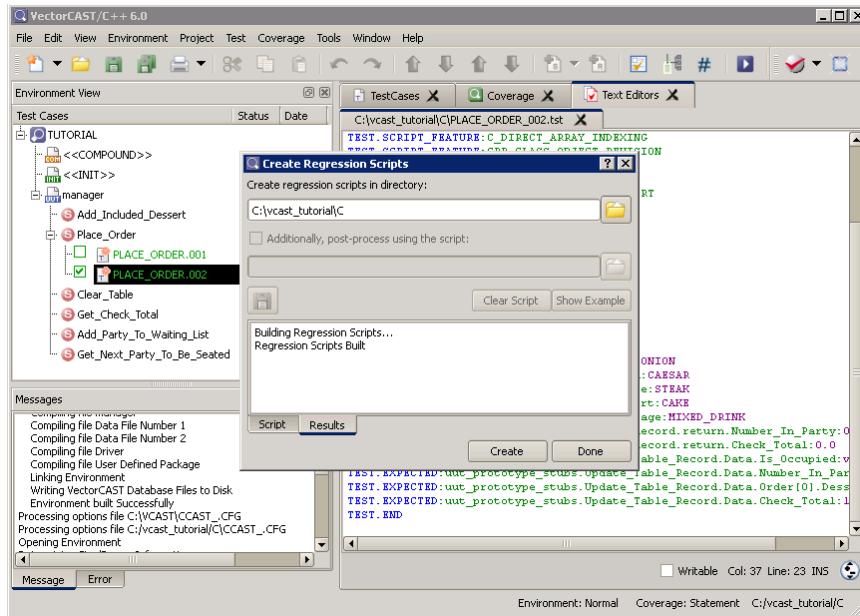


2. In the first edit box, click the **Browse** button and navigate to any directory you choose. For the purposes of this tutorial, we will specify the working directory as the destination for the regression scripts.



3. Click **Create**.

VectorCAST creates the three regression scripts in the location you specified.



- Click **Done** to close the dialog.

Now that you have the regression scripts, you can delete this environment and easily rebuild it at a later time simply by executing the shell script.

Tutorial Summary

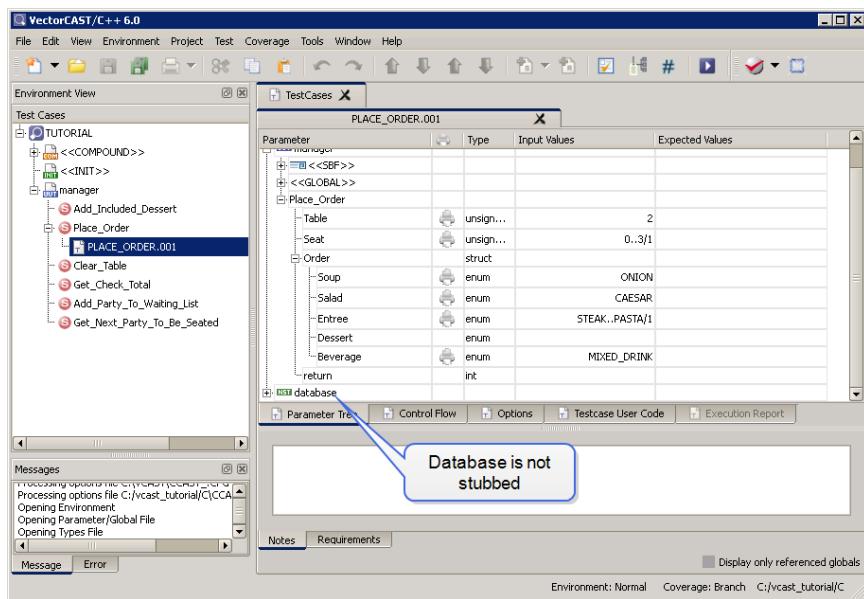
You have now completed the Basic Tutorial. You:

- > Built a test environment named **TUTORIAL** for testing a UUT named **manager**
- > Defined and executed a test case named **PLACE_ORDER.001**
- > Viewed source coverage information
- > Viewed the Test Case Management Report and the Aggregate Coverage Report for your test environment
- > Duplicated a test case
- > Stubbed a function in the UUT during test execution
- > Created regression scripts for this environment, which include the shell script, test script, and environment script

Troubleshooting

Problem: Dependents are not stubbed

If you do not see **Get_Table_Record** (only <<GLOBAL>> and **Table_Data**, as shown below), then you did not stub **database**.



As a result, the test harness is using the actual unit (database), which means you cannot specify any input or expected values for subprogram Get_Table_Record.

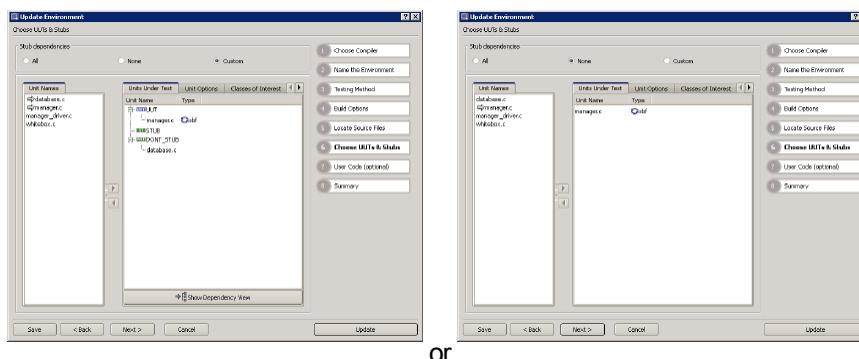
To solve this problem, you can update the environment to change database from not stubbed to stub:

1. Select **Environment => Update Environment**.

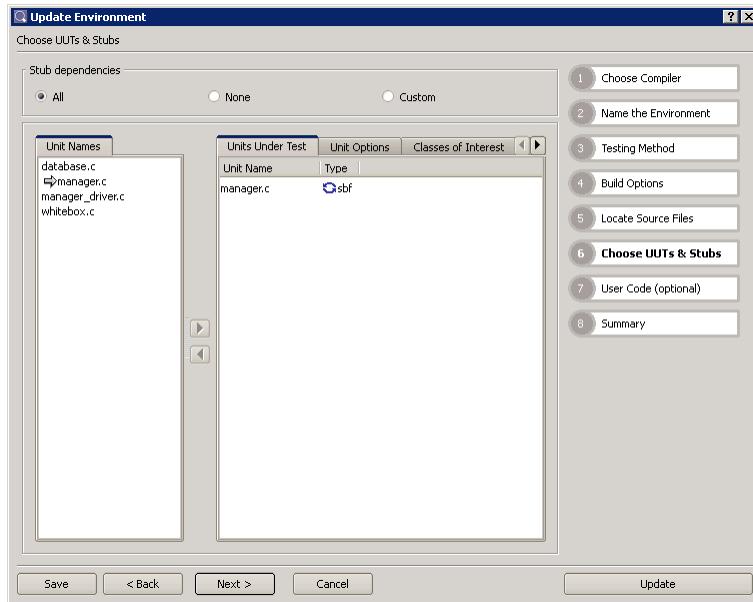
A dialog appears that looks similar to the Create New Environment dialog.

2. Click on **Choose UUTs & Stubs**.

Depending on whether you chose “Not Stubbed” when cycling through the types of stubbing or you selected the stub “None” radio button along the top, the Wizard looks like the following:



3. Click the radio button **All** under **Stub dependencies** to change **database** to be stubbed.



(Alternatively, you could have clicked **Custom**, then the toggle button  and then cycled the unit **database** until its type was **stub (by prototype)**).

Unit Name	Type
manager.c	sbf
database.c	stub (by prototype)

4. Click **Update**.

The environment rebuilds; **database** is now stubbed.

Problem: Input values entered instead of expected values

If you enter expected values into Input Test Data fields, you will see the test-case listing as shown below. Note that there is no information under **Expected Test Data**.

Input Test Data		
UUT: manager.c		
Subprogram: Place_Order		
Table	2	
Seat	0	
Order		
Soup	ONION	
Salad	CAESAR	
Entree	STEAK	
Beverage	MIXED_DRINK	
Stubbed Subprograms:		
Unit: manager.c		
Subprogram: Get_Table_Record		
return		
Number_In_Party	0	
Check_Total	0.0	
Subprogram: Update_Table_Record		
Data		
Is_Occupied	v_true	
Number_In_Party	1	
Check_Total	VARY FROM:12.0 TO:16.0 BY:1.0	
Expected Test Data		
Test Case / Parameter Input User Code		

To solve this problem:

1. Select **File => Close** to close the test-case data report.
2. Open test case **PLACE_ORDER.001** for editing.
3. Delete the values **v_true**, **1**, and **12..16** from the **Input Values** column and enter them into the **Expected Values** column.

Parameter	Type	Input Values	Expected Values
Place_Order			
Table	unsigned short	2	
Seat	unsigned short	0	
Order	struct		
return	int		
Stubbed Subprograms			
Get_Table_Record			
Update_Table_Record			
Table	unsigned short		
Data	struct		
Is_Occupied	enum	v_true	1
Number_In_Party	unsigned short		
Designator	char		
Wait_Person	char		

Multiple UUT Tutorial

This tutorial demonstrates how to use VectorCAST's multiple UUT feature to conduct integration testing.

When testing more than one UUT in an environment, it is typical that the units interact. VectorCAST's multiple UUT (multi UUT) feature enables you to do this.

In VectorCAST, there are two types of test cases: simple test cases and compound test cases. In the Basic Tutorial, you created and used a simple test case, which corresponded to a single invocation of a UUT. A compound test case is a collection of simple test cases that invokes a UUT in different contexts. The *data is persistent* across these invocations. Data persistence is very important when doing integration testing.

This tutorial will take you through the steps of building a compound test case to test the interaction between the unit **manager** and the unit **database**. The unit **manager** is a simple database in which actual data is stored in data objects defined within the unit. You are guided through the steps of building a compound test case to write data into this database. You then use the same compound test case to retrieve data from the database to verify it.

It is recommended that you review the relevant source listings in Appendix A, "Tutorial Source Code Listings" on page 311 before proceeding.

What You Will Accomplish

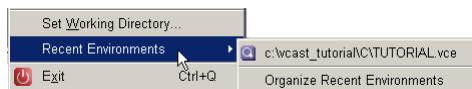
In this tutorial you will:

- > Build a multi UUT environment by updating the environment you built in the Basic Tutorial
- > Add a test case to unit **database**
- > Use the simple test cases to create a compound case
- > Verify the data flow between UUTs **manager** and **database**
- > Set processing iterations across ranges of input data
- > Build and execute a compound test case

Building a Multiple UUT Environment

In this section, you will modify for integration testing the environment you built in the Basic Tutorial.

1. Start VectorCAST if it is not running.
If necessary, refer to ."Starting VectorCAST" on page 8
2. Select **File => Recent Environments**, and then select the environment you created in the first tutorial (named **TUTORIAL.vce** by default):



The test cases present in this environment were created on the basis of **database** being a stubbed dependent of **manager**. However, in your new environment, you want **database** to be a

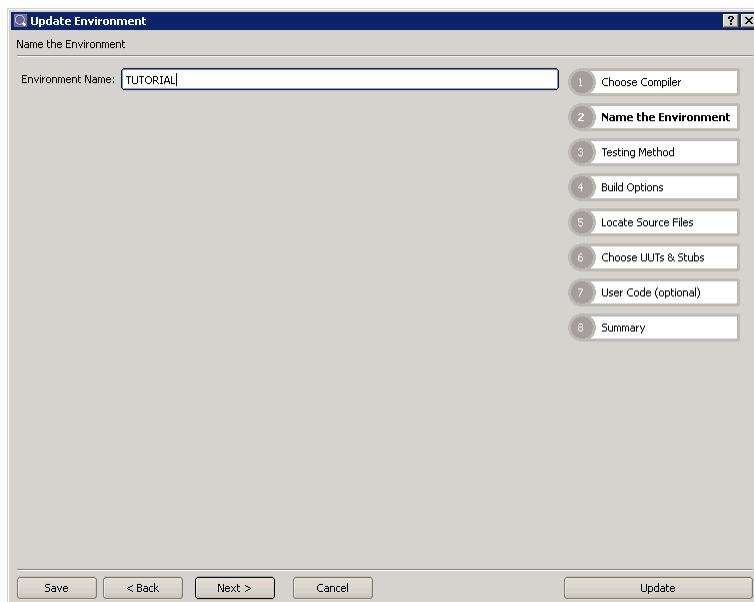
second UUT. No problem! As you will see in the next section, VectorCAST converts the values in a stubbed unit to corresponding values in a stubbed-by-function unit.

Updating the Environment

You will now change **database** from being a stubbed dependent in your test environment to being to a UUT:

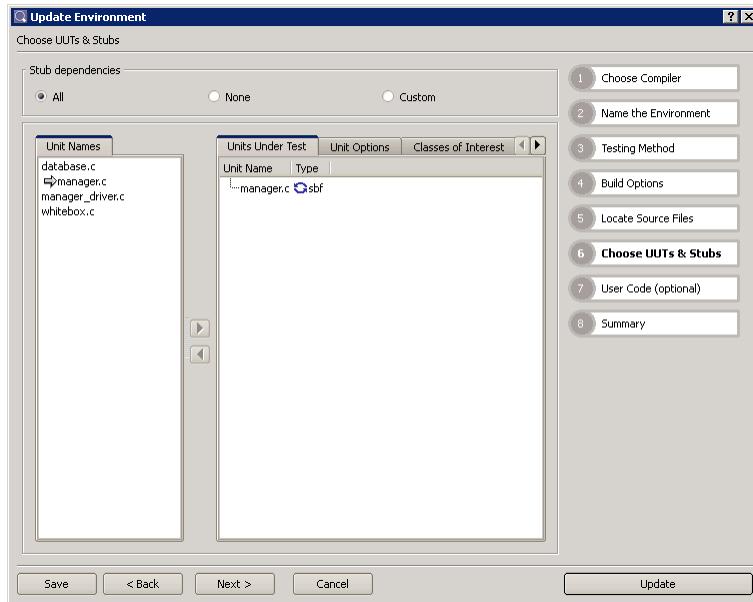
1. Select **Environment => Update Environment**.

The Update Environment wizard appears:



This wizard is similar to the Create Environment wizard you used in the Basic Tutorial to create the original environment. There are no red outlines because there are no errors or missing items of information.

2. Click **Choose UUTs & Stubs** , step 6.

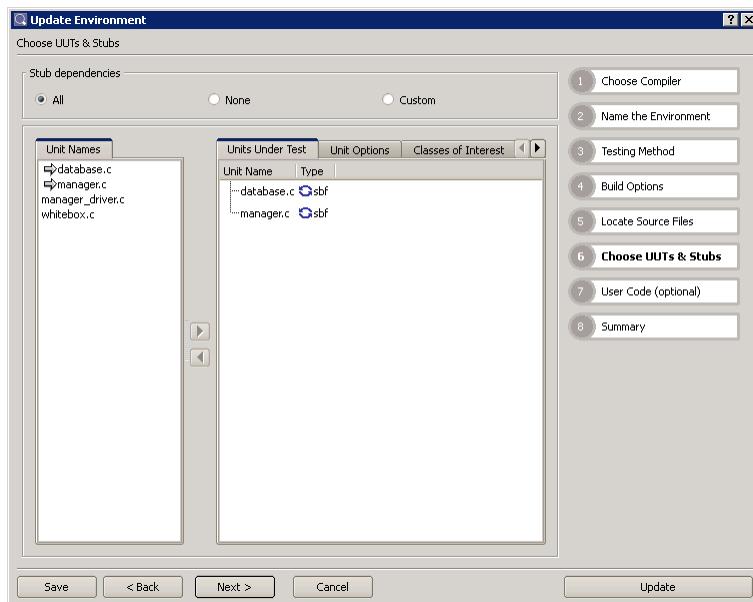


3. In the **Unit Names** pane, double-click **database.c**.



Note: Double-clicking a name in the Unit Names pane is a shortcut for selecting the name and then clicking the right-arrow ().

Both **database** and **manager** are now listed as Units Under Test with Stub-by-function (SBF) enabled:



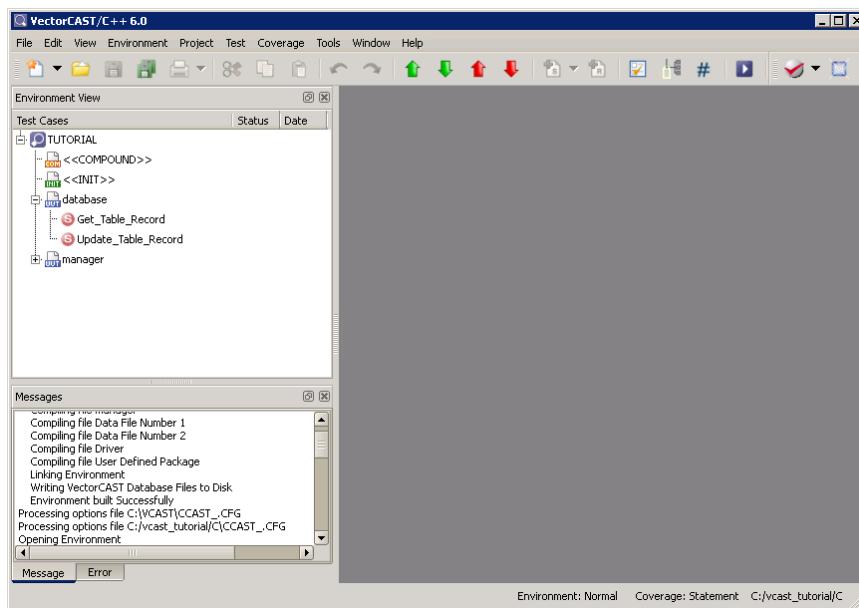
4. Click **Update**.

5. As the environment updates to include two UUTs, a series of messages is displayed in the Message window. A message appears confirming that the update has completed:



6. Click **OK**.

The environment is opened:



Note in the Environment View that **database** is now listed as a UUT, and also that it has two subprograms: **Get_Table_Record** and **Update_Table_Record**.

Next, you will edit one of the test cases in Place_Order and create a simple **database** test case, and then combine these two simple cases to create a compound case.

Building a Compound Test Case

You now have two UUTs to use toward creating a compound test case.

You create a compound test case by creating a separate test case for each subprogram to be included in the test and then combining these test cases into a compound test case.

To create the simple test cases, you use the same procedure you used in the Basic Tutorial to create the test case (PLACE_ORDER.001) that you deleted for the purposes of this tutorial.

In this section, you will:

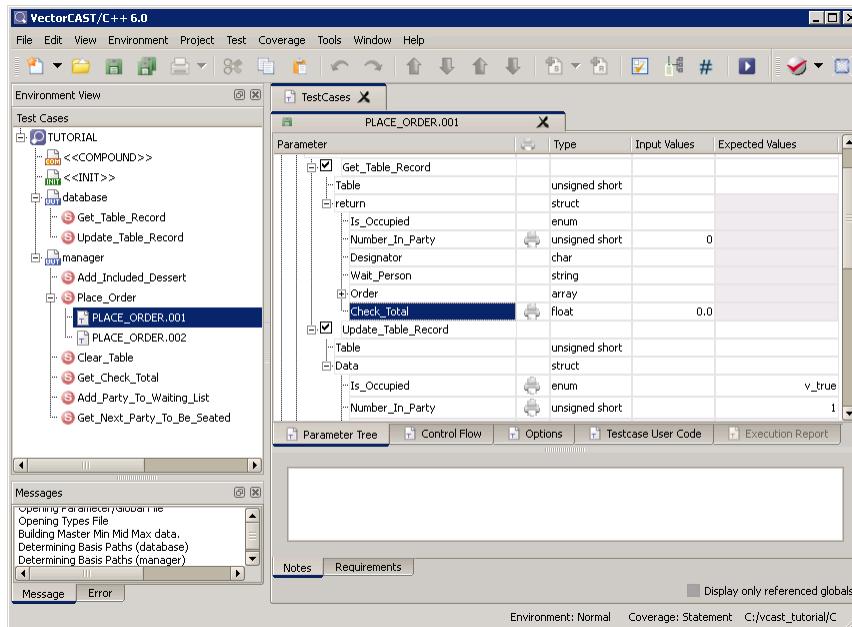
- > Turn off stubbing-by-function in the test case PLACE_ORDER.001. We want to call the real code for the subprograms in the unit database in this tutorial.
- > Create a simple test case for subprogram **Get_Table_Record**, which reads data from the database. This test case will be used to verify that **Place_Order** is updating the database properly.
- > Place these test cases into a compound test case

Edit the test case PLACE_ORDER.001

To turn off stubbing-by-function in PLACE_ORDER.001:

1. In the Environment View, expand **manager** into its subprograms, and then expand **Place_Order** into its test cases.
2. Double-click **PLACE_ORDER.001**.

The test case PLACE_ORDER.001 displays in the Test Case Editor.



Note that after the Environment Update, the subprograms **Get_Table_Record** and **Update_Table_Record** are stubbed (by-function). But because this is an integration test, we do not want these subprograms stubbed in order to verify the “real” code for these subprograms.

3. Uncheck the checkbox next to **Get_Table_Record** under the <>SBF>> node.
4. Uncheck the checkbox next to **Update_Table_Record** as well.

Parameter	Type	Input Values	Expected Values
USER_GLOBALS_VCAST			
UUT database			
<<SBF>>			
Get_Table_Record			
Update_Table_Record			
<<GLOBAL>>			
UUT manager			
<<SBF>>			
<<GLOBAL>>			
Place_Order			
Table	unsigned short	2	
Seat	unsigned short	0	
Order	struct		
Soup	enum	ONION	

The test case already sets the Entrée, Soup, Salad, and Beverage. This combination yields an included dessert of Pie, so we will expect PIE when later verifying the Order.

- Save the test case by clicking the **Save** button  on the toolbar.

Because you will be making your test case part of a compound test case, *and* you only want to use this test within a compound case, you need to designate it as Compound Only. Compound Only test cases can only be executed from within a compound test. They cannot be executed individually or as part of a batch.

In some cases, you will want to use simple test cases both individually and as part of a compound case. In these instances, it is not necessary to apply the Compound Only attribute to the simple cases.

- Right-click **PLACE_ORDER.001** and select **Compound Only**.

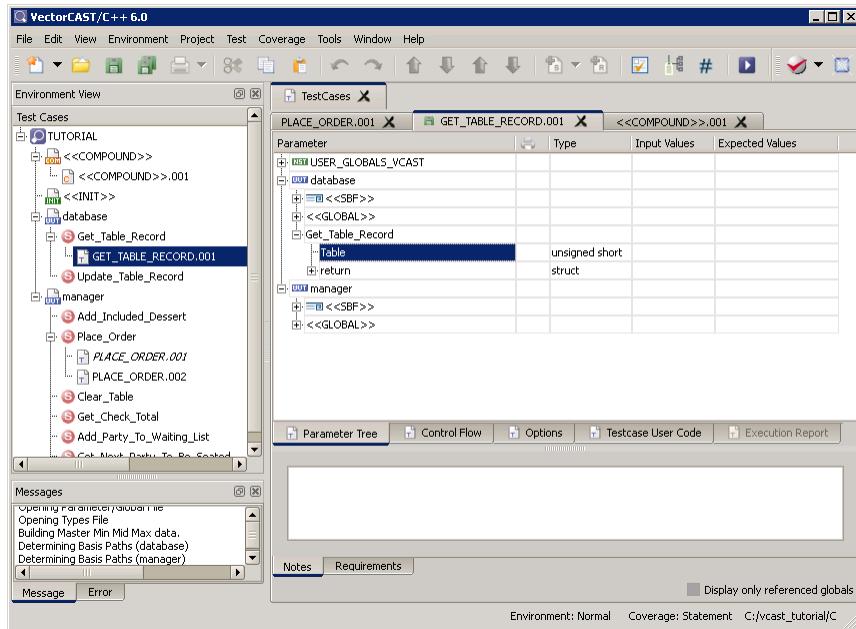
The test case typeface changes to *italic*, signifying that this test case has been designated Compound Only.

Create a Test Case for the Get_Table_Record Function

In this section, you will create a test case for the function **Get_Table_Record**. This function retrieves data from the database and verifies it against a set of expected values.

- In the Environment View, expand **database** into its subprograms (if not already expanded).
- Right-click **Get_Table_Record** and select **Insert Test Case**.

The test case GET_TABLE_RECORD.001 opens in the Test Case Editor:



- Keeping with our example situation, enter **2** as the input value for **Table**.



Note: Be sure you enter this value into the Input Values column, not into the Expected Values column. You will enter the expected values for this case in the next step.

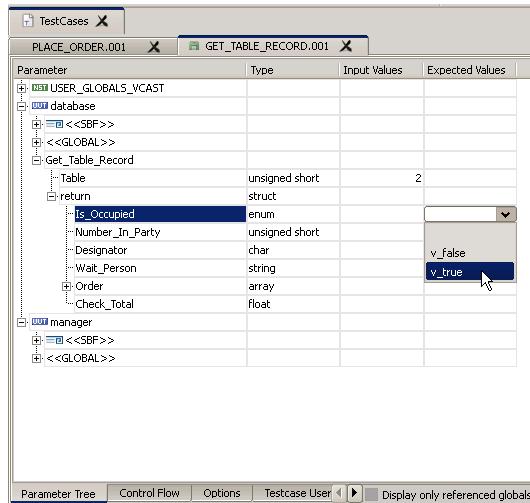
- Expand **return** (click \oplus), then enter the following **Expected Values**:

Is_Occupied **v_true**

Number_In_Party **1**



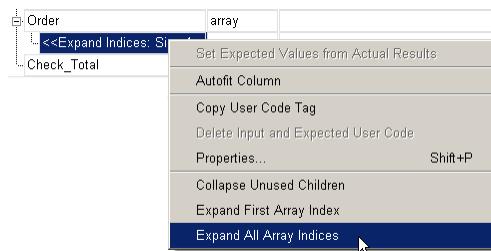
Note: Make sure you enter these two values in their respective **Expected Values** fields.



5. **Expand Order.**

Order is an array of structures of size 4.

6. To view the array indices of **Order**, right-click on <<Expand Indices: Size 4>> and select **Expand All Array Indices**.



The array expands, displaying the four array elements:

Parameter	Type	Input Values	Expected Values
USER_GLOBALS_VCAST			
database			
<<GLOBAL>>			
Get_Table_Record			
Table	unsigned short	2	
return	struct		
Is_Occupied	enum		v_true
Number_In_Party	unsigned short		1
Designator	char		
Wait_Person	string		
Order	array		
<<Expand Indices: Size 4>>			
Order[0]	struct		
Order[1]	struct		
Order[2]	struct		
Order[3]	struct		
Check_Total	float		
manager			
<<GLOBAL>>			

In this example, you want to verify input values written to the application database. The data consists of an order taken for seat 0 at table 2.

7. Expand **Order[0]**.

Order[0] corresponds to seat 0. The expected values you specify for Order[0] must match the input values you previously specified in PLACE_ORDER.001.

Important: If you use any other index, the expected results will not match.

8. Assign the following **Expected Values** to the appropriate fields of **Order[0]**:

Soup	ONION
Salad	CAESAR
Entree	STEAK
Dessert	PIE
Beverage	MIXED_DRINK

 **Note:** Make sure you enter these values into the **Expected Values** column:

The screenshot shows the VectorCAST TestCases interface with the 'PLACE_ORDER.001' test case selected. In the Parameter Tree, under the 'Get_Table_Record' section, there is a 'Order' node which is expanded to show 'Order[0]', 'Order[1]', 'Order[2]', and 'Order[3]'. The 'Order[0]' node has several sub-fields: 'Soup' (enum, value: ONION), 'Salad' (enum, value: CAESAR), 'Entree' (enum, value: STEAK), 'Dessert' (enum, value: PIE), and 'Beverage' (enum). A dropdown menu is open over the 'Beverage' field, listing five options: MIXED_DRINK, NO_BEVERAGE, BEER, WINE, and SODA. The 'MIXED_DRINK' option is highlighted. The 'Order[1]', 'Order[2]', and 'Order[3]' nodes are collapsed.

- Click to save.

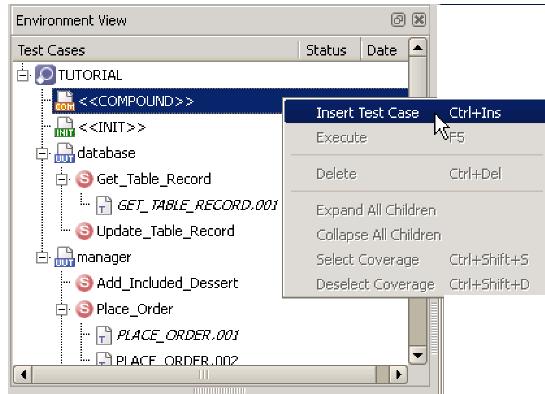
Note: Order[1], Order[2], and Order[3] are not populated with data. Populating a single Order record is sufficient to verify the operation of the two subprograms under test.

- Right-click **GET_TABLE_RECORD.001** in the Environment View, then select **Compound Only**. You have two simple test cases designated Compound Only. The first (PLACE_ORDER.001) takes a dinner order for table 2, seat 0. The order is then stored in the application database by means of `Place_Order` calling `Update_Table_Record`. The second simple test case (GET_TABLE_RECORD.001) retrieves data from the database for verification.

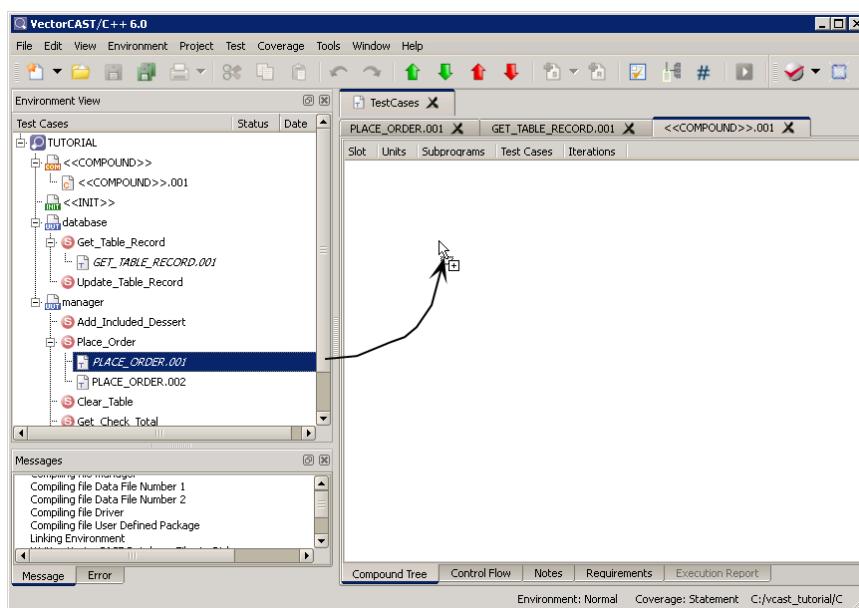
Creating a Compound Test Case

If you were to execute your test cases individually, they would fail, because the test harness would be called separately for each test case, and the data would not be retained in the data structure between the two calls. In order for your test cases to work, they must be combined into a compound case.

1. In the Environment View, right-click <<COMPOUND>> and select **Insert Test Case**.
A test case named <<COMPOUND>>.001 appears under <<COMPOUND>>, and is selected (highlighted) by default.



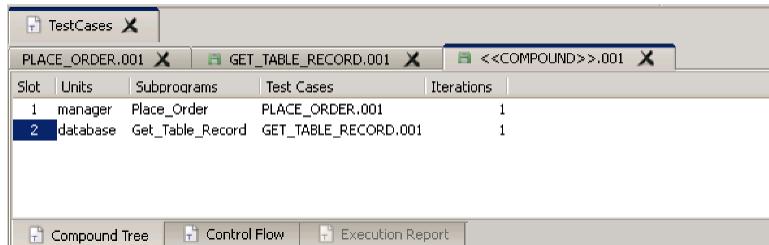
2. Click and hold the cursor on **PLACE_ORDER.001** in the Environment View; drag until the symbol appears in the Test Case Editor; release.



An entry appears in the Test Case Editor (slot 1) as a member of test case <<COMPOUND>>.001.

3. Do the same with test case **GET_TABLE_RECORD.001**.

Your compound test case now has two slots: PLACE_ORDER.001, and GET_TABLE_RECORD.001:



4. Click **Save**.

You are now ready to execute your compound test case.

5. In the Environment View, select <<COMPOUND>>.001, then click the Execute button on the toolbar.

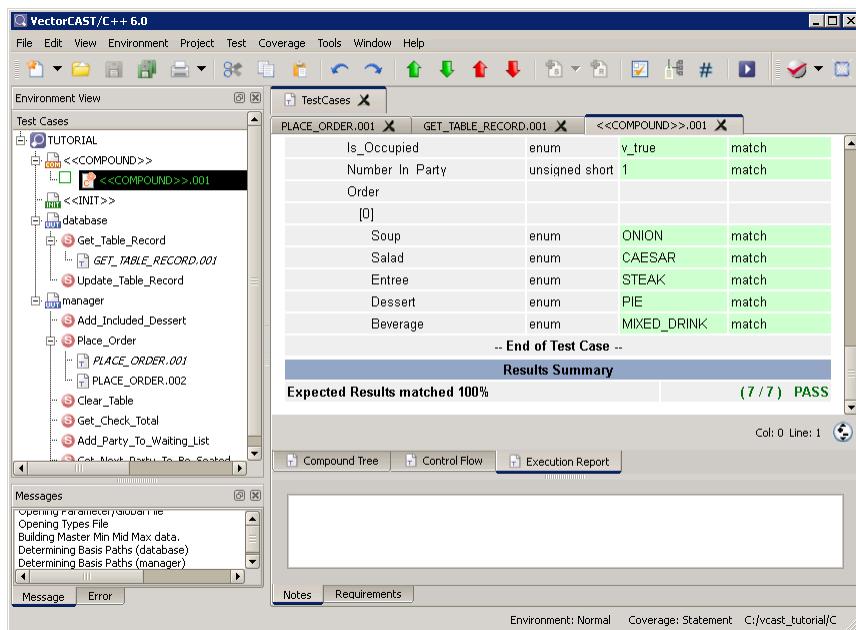
An execution report for the compound test case appears in the Report Viewer.



Note: If your compound test case fails, make sure PLACE_ORDER.001 uses parameter values table 2, seat 0, and that GET_TABLE_RECORD.001 uses table 2 and the proper expected values in Order[0].

6. Scroll down the report.

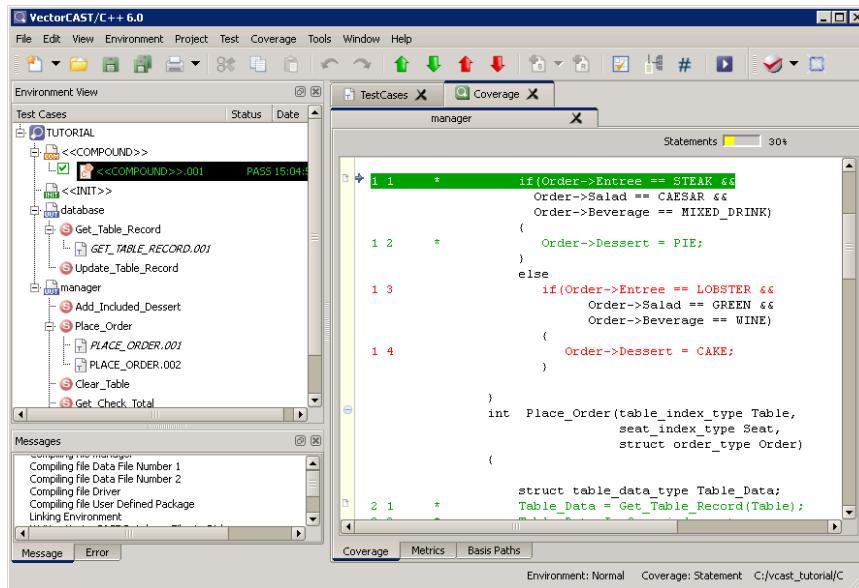
Note that the expected values you specified were matched against corresponding actual values, verifying the data flow from **manager** to **database**.



7. Click the **checkbox** preceding <<COMPOUND>>.001 in the Environment View:



The tab for **manager** appears in the Coverage Viewer.



In the Coverage Viewer, note that the statement assigning the value PIE to Order-> Dessert is green, indicating that it was covered by test execution.

Initializing Branch Coverage

To find out which branches are covered by executing this compound test case:

1. Choose **Coverage => Initialize => Branch**.

The status bar shows “Coverage: Branch” and the checkbox next to <<COMPOUND>>.001 is removed, because the statement results were made invalid during coverage initialization. To see branch coverage achieved, you must execute the test case again.

Note that the *execution results* were not made invalid during coverage initialization.

2. Select <<COMPOUND>>.001 in the Environment View and click the **Execute** button  in the toolbar.
3. Click the **checkbox** preceding <<COMPOUND>>.001 in the Environment View:



4. Scroll to the first covered line of **Place_Order**.

In the Coverage Viewer, the instances of the '(T)' symbol indicate that two `Place_Order` branches were covered: the first at the subprogram entry point; the second at the STEAK branch in the `switch` statement.

VectorCAST allows you to build more complex tests by linking multiple simple test cases together. This feature allows you to test threads of execution while maintaining the data common to the various units and subprograms.

Adding Test Iterations for Fuller Coverage

You can reach fuller coverage of `manager` by having each seat at table 2 order a different entrée while keeping the other order items constant. Varying the Entrée value ensures that each branch in the `switch` statement will be exercised by the test.

In this section, you will modify test case `PLACE_ORDER.001` to iterate over the 4 seats at table 2 while iterating over the four entrées. The net effect will be to assign a different entrée to each seat.

1. In the Environment View, double-click **`PLACE_ORDER.001`**.

The test case `PLACE_ORDER.001` opens in the Test Case Editor. If it does not appear, click the **Parameter Tree** tab at the bottom of the Test Case Editor.

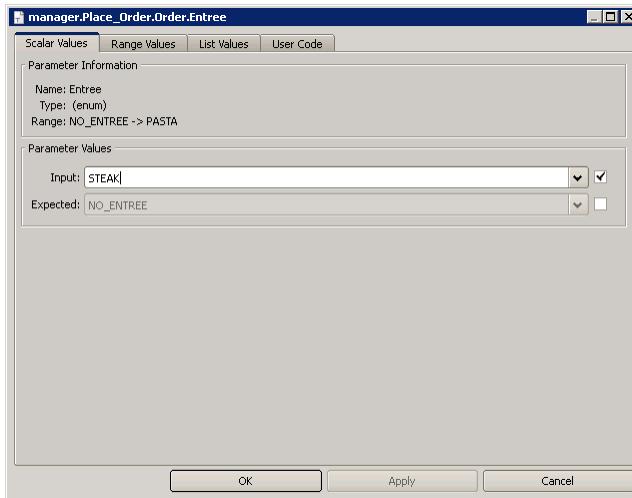
2. Enter **0..3** as the input value for **Seat**; press **Enter**.

Your entry changes to `0..3/1`. The `/1` indicates that the input value for Seat will iterate over its range with a delta of 1. (It is possible to specify a delta other than 1.)

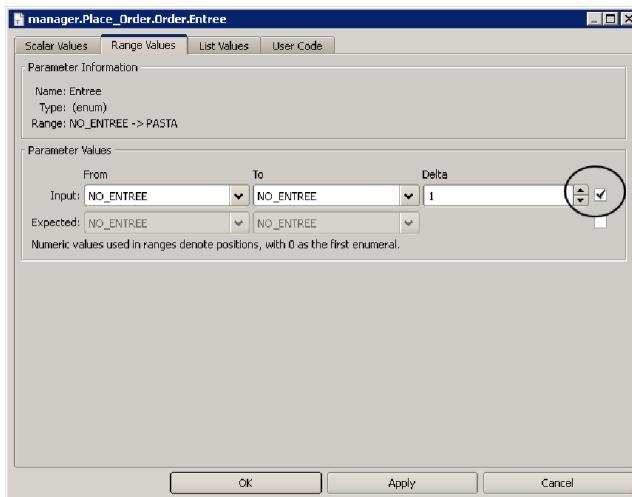
Parameter	Type	Input Values	Expected Values
+ NET_USER_GLOBALS_VCAST			
+ UUT database			
+ <<SBF>>			
+ Get_Table_Record			
+ Update_Table_Record			
+ <<GLOBAL>>			
- UUT manager			
+ <<SBF>>			
+ <<GLOBAL>>			
- Place_Order			
+ Table	unsigned short	2	
+ Seat	unsigned short	0..3/1	
+ Order	struct		
+ Soup	enum	ONION	

3. Double-click **Entrée**.

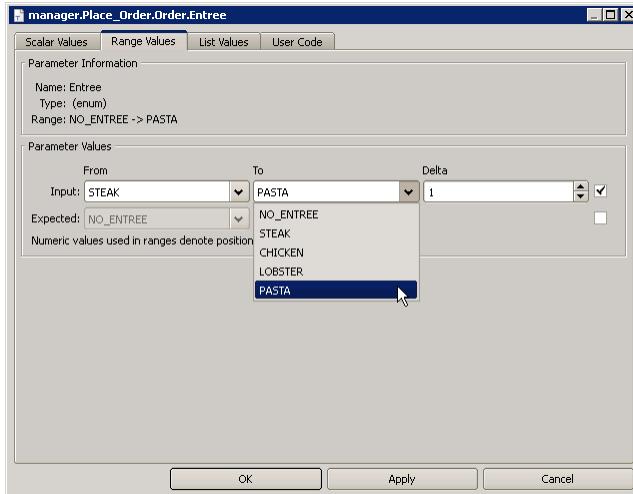
A dialog appears for setting parameter specifications:



4. Click the **Range Values** tab.
5. Enable the **Input** fields by clicking the checkbox at the far right:



6. Select **Steak** for **From**; **Pasta** for **To**; leave **Delta** at **1**:



Note: You can also enter a range of values by typing it directly into the appropriate Input Values field, in the appropriate format.

7. Click **OK**.

Parameter	Type	Input Values	Expected Values
<<GLOBAL>>			
UUT manager			
<<SBF>>			
<<GLOBAL>>			
Place_Order			
-- Table	unsigned short	2	
-- Seat	unsigned short	0..3/1	
Order	struct		
Soup	enum	ONION	
Salad	enum	CAESAR	
Entree	enum	STEAK..PASTA/1	
Dessert	enum		
Beverage	enum	MIXED_DRINK	

8. Click **Save**.

You have now assigned input ranges to two parameters. By default, when two parameters are assigned input ranges, they will iterate in *parallel*, which is what you want to occur in this case. (There is a setting to make them iterate in combination.)

These two parameters will iterate with the following value pairs:

Seat = 0 Entree = Steak

Seat = 1 Entree = Chicken

Seat = 2 Entree = Lobster

Seat = 3 Entree = Pasta

Given this information, you can specify expected values in your compound test case for each seat.

9. Click the tab for test case **GET_TABLE_RECORD.001** at the top of the Test Case Editor. The parameter tree for GET_TABLE_RECORD.001 comes to the front. You need to edit this tree to reflect the expected results for your compound test. You now need to specify expected values for each of the other three seats at table 2.
10. You have already specified values for Order[0], so move on to Order[1].
11. Expand **Order[1]**. (If you have recently re-opened the TUTORIAL environment, then the unused array elements have been collapsed. If Order[1] is not visible, simply right-click on <>Expand Indices: Size 4>> and choose **Expand All Array Indices**.)

Order[1] maps to seat 1. Specify the following expected values for seat 1:

Soup	ONION
Salad	CAESAR
Entree	CHICKEN
Dessert	NO_DESSERT (see Note)
Beverage	MIXED_DRINK



Note: For Order[1], Order[2], and Order[3], Place_Order calls Add_Included_Dessert with a soup, salad, and entrée combination that does not qualify for Pie, so we expect NO_DESSERT.

-
12. Expand **Order[2]**. Enter the following expected values for seat 2:
- | | |
|----------|--------------------|
| Soup | ONION |
| Salad | CAESAR |
| Entree | LOBSTER |
| Dessert | NO_DESSERT |
| Beverage | MIXED_DRINK |

13. Expand **Order[3]**. Enter the following expected values for seat 3:

Soup	ONION
Salad	CAESAR
Entree	PASTA
Dessert	NO_DESSERT
Beverage	MIXED_DRINK



Note: Be sure you have populated the Expected Values column for Order, and not the Input Values column.

You now have Order data for each seat at table 2. Note that these orders are identical in each case except for the Entree and the Dessert. Only the Entree variable is needed to exercise each branch in the **switch** statement.

Parameter		Type	Input Values	Expected Values
	Order[1]	struct		
	--Soup	enum		ONION
	--Salad	enum		CAESAR
	--Entree	enum		CHICKEN
	--Dessert	enum		NO_DESSERT
	--Beverage	enum		MIXED_DRINK
	Order[2]	struct		
	--Soup	enum		ONION
	--Salad	enum		CAESAR
	--Entree	enum		LOBSTER
	--Dessert	enum		NO_DESSERT
	--Beverage	enum		MIXED_DRINK
	Order[3]	struct		
	--Soup	enum		ONION
	--Salad	enum		CAESAR

You must now change the number of occupied seats at table 2 to reflect the current test case.

14. Scroll up to **Number_In_Party** and change the expected value to **4**.
15. Click **Save**.
16. Select **<<COMPOUND>>.001** in the Environment View; click to execute. **<<COMPOUND>>.001** turns green; an Execution Report for the compound test appears in the Test Case Editor. Your test has passed.
17. Click the green down-arrow or scroll down a bit to view the green-coded results for each Order field:

Is_Occupied	enum	v_true	match
Number_In_Party	unsigned short	4	match
Order			
[0]			
Soup	enum	ONION	match
Salad	enum	CAESAR	match
Entree	enum	STEAK	match
Dessert	enum	PIE	match
Beverage	enum	MIXED_DRINK	match
[1]			
Soup	enum	ONION	match
Salad	enum	CAESAR	match
Entree	enum	CHICKEN	match

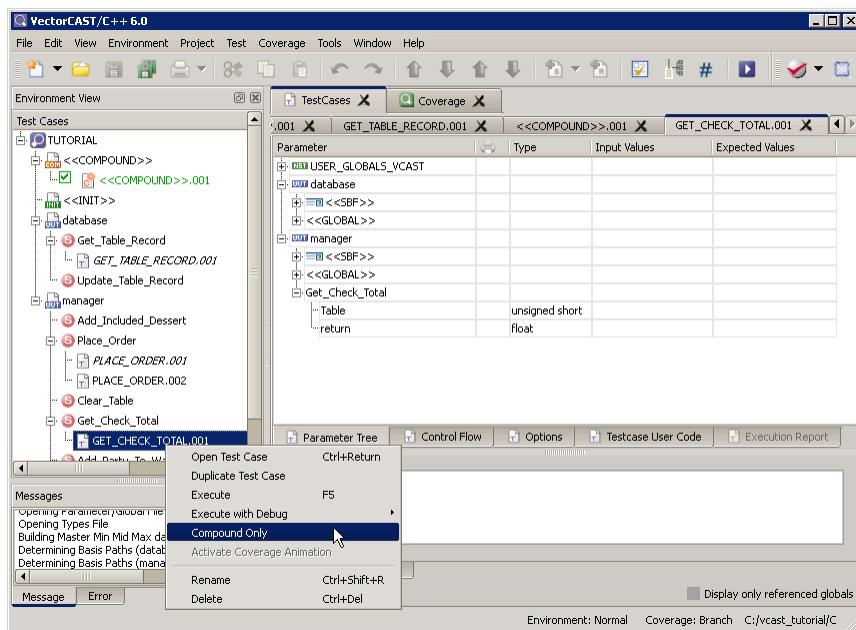
Adding Test Cases to a Compound Test

In this section, you will verify the check total for the four entrée orders at table 2 by adding a third test case to your compound case.



Note: The amount of instruction given in this section will be a little less than in the previous sections.

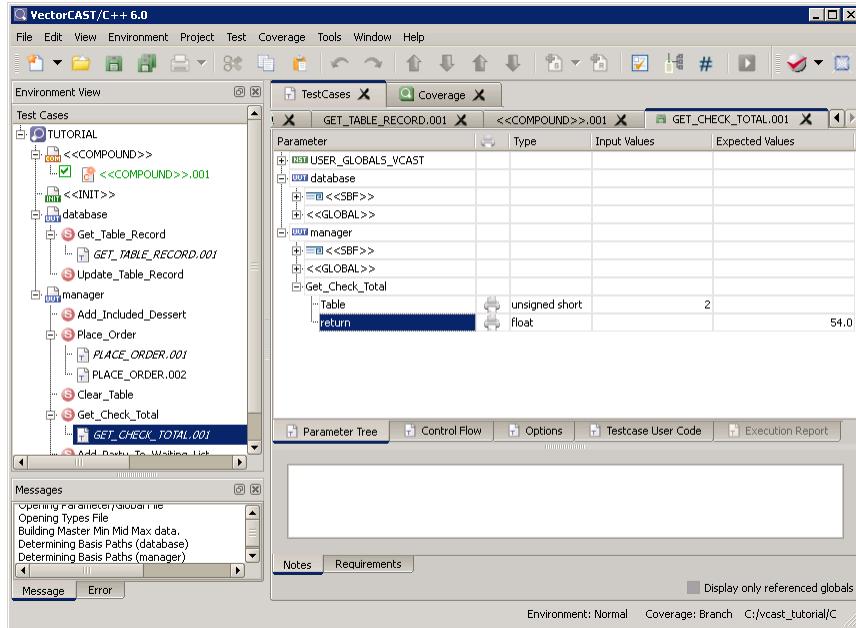
- Under the **manager** unit, insert a test case for **Get_Check_Total** and make the test case **Compound Only**.



- Enter **2** as the *input* value (*not* expected value) for **Table**.

On the basis of the entrée prices coded into the application, you expect the check to total \$54 (for one of each entrée).

- Enter **54** as the expected value for **return**.



4. Save.

You can now add your new test case to the compound case.

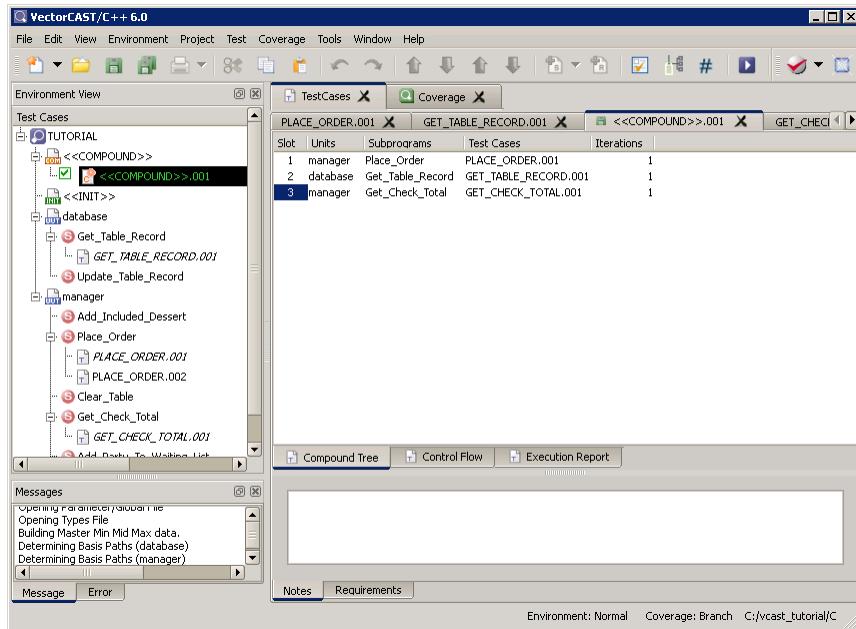
5. Double-click **<<COMPOUND>>.001** in the Environment View.

6. Click the **Compound Tree** tab at the bottom of the Test Case Editor.

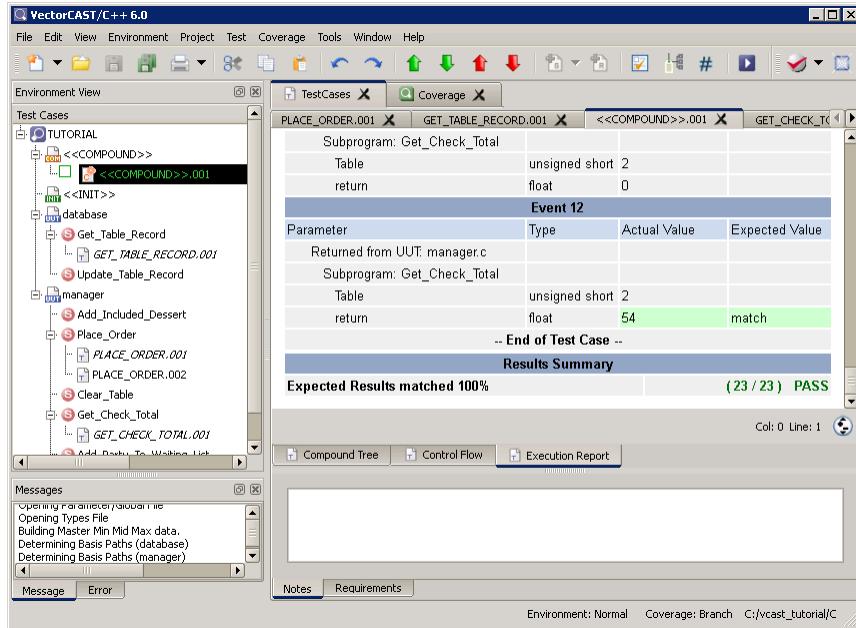
The Test Case Editor displays the slots in test case **<<COMPOUND>>.001**.

7. Add test case **GET_CHECK_TOTAL.001** to the compound by dragging and dropping it as the last slot.

GET_CHECK_TOTAL.001 displays as slot 3 in your compound test case:

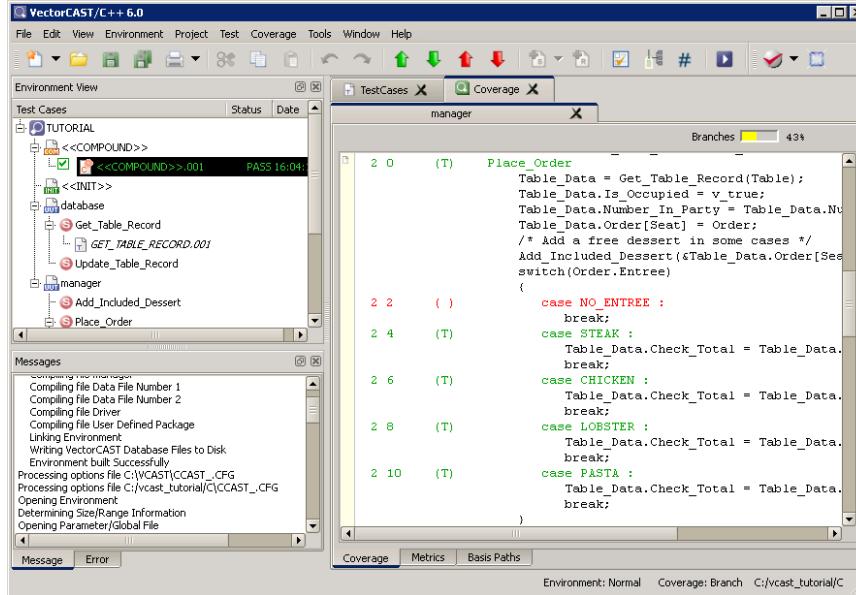


8. Save.
9. Select <>COMPOUND>>.001 in the Environment View, then click to execute it. <>COMPOUND>>.001 turns green in the Environment View; and an execution report displays in the Test Case Editor.
10. Click the green down-arrow on the toolbar, or scroll down the report until you come to the green-shaded results section. You can see that the check total for table 2 was \$54.



11. To view the coverage data, click the green **checkbox** preceding <<COMPOUND>>.001 in the Environment View.

The Coverage Viewer opens.



Notice from the green coloring that by iterating over four entrées, you exercised four of the five **switch** branches. Notice from the status bar that total branch coverage was 10 of 23, or 43%. At this point, you might want to initialize a different type of coverage. In addition, you might want to try adding a test case for **clear_Table** (with **Table** also set at **2**).

Exporting a Test Script

1. Click on the top level of the environment, **TUTORIAL**, in the Environment View.
2. Select **Test => Scripting => Export Script...**, and give the test script the name **TUTORIAL_MULTI.tst**.
3. Click **Save** to create the test script.

Tutorial Summary

This tutorial built on the environment you created in the Basic Tutorial. You changed one of the dependent units from a smart stub into a UUT. You then added test cases to both units, and combined these test cases into a Compound test case, which you used to test the data flow from `manager` to `database`. When test cases are combined into a Compound test case, the data remains persistent between them.

In sum, you:

- > Built a multi UUT environment by updating the environment you built in the Basic Tutorial
- > Turned off stubbing-by-function in the test case from `manager`
- > Added a test case to `database`
- > Used these simple test cases to create a compound case
- > Verified the data flow between UUTs `manager` and `database`
- > Set up range iterations for a scalar parameter and an enumeral
- > Added a test case to an existing compound case

C++ Tutorials

Introduction

The tutorials in this chapter demonstrate how to use VectorCAST/C++ to unit-test an application written in the C++ programming language.

For a comprehensive explanation of the VectorCAST/C++ features, refer to the *VectorCAST/C++ User's Guide*.

The modules you will test in the first two tutorials are components of a simple order-management application for restaurants. The source listings for this application are available in Appendix A, "Tutorial Source Code Listings" on page 311. It is recommended that you at least scan through these listings before proceeding with the tutorials.

You should run the tutorials in this chapter in the order presented.



Tip: You can stop a tutorial at any point and return later to where you left off. Each tutorial session is automatically saved. To return to where you left off, simply restart VectorCAST and the use **File => Recent Environments** to reopen the tutorial environment.

Basic Tutorial

The Basic Tutorial will take you through the entire process of using VectorCAST/C++ to test a software unit.

What You Will Accomplish

In this tutorial, you will:

- > Build a test environment (which includes an executable test harness)
- > Create test cases for a unit under test (UUT)
- > Execute tests on a UUT
- > Analyze code coverage on a UUT
- > Stub a function in the UUT



Note: User entries and selections are represented in this tutorial in bold. For example: "Enter **Demo**, then click **OK**".

Preparing to Run the Tutorial

Before you can run this tutorial, you need to decide on a working directory and copy into this directory the Tutorial files from the VectorCAST installation directory. On Windows, a good choice is the default working directory, named **Environments**, located in the VectorCAST installation directory. On UNIX, it often is desirable to create your own working directory.

For efficiency, this tutorial assumes a working directory named **vcast_tutorial** located at the top level in your file tree, and a sub-directory named **cpp**. You can, however, locate this directory any place you want, and make the necessary translations in the text as you progress through the tutorial. Also for

efficiency, this tutorial assumes you will be running the tutorial in a Windows environment, using Windows syntax for file paths and things specific to an operating system (Windows, Solaris, and UNIX). Again, you should have no problems making the necessary translations.

The files you need to copy are located in the directory in which you installed VectorCAST. In Windows, these files are located in %VECTORCAST_DIR%\Tutorial\cpp by default.

In Windows, enter:

```
C:>mkdir vcast_tutorial\CPP
C:>copy %VECTORCAST_DIR%\Tutorial\cpp vcast_tutorial\CPP
```

In UNIX, enter:

```
$ mkdir vcast_tutorial/CPP
$ CP $VECTORCAST_DIR/TUTORIAL/CPP/* VCAST_TUTORIAL/CPP
```

Building an Executable Test Environment

In this section, you will use the VectorCAST/C++ tool to create an executable test environment. The sample UUT you will use is named **manager**.

You will:

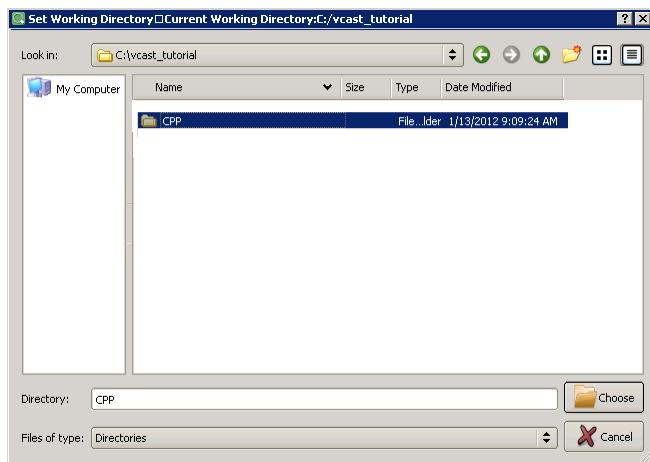
- > Specify a working directory
- > Specify an environment type (C/C++)
- > Specify a compiler
- > Name the environment to be created
- > Turn on whitebox
- > Specify the source files to be included in the environment
- > Designate the UUT and any stubs (as well as the method of stubbing)
- > View a summary of your specifications
- > Build the test environment

Specifying a working directory

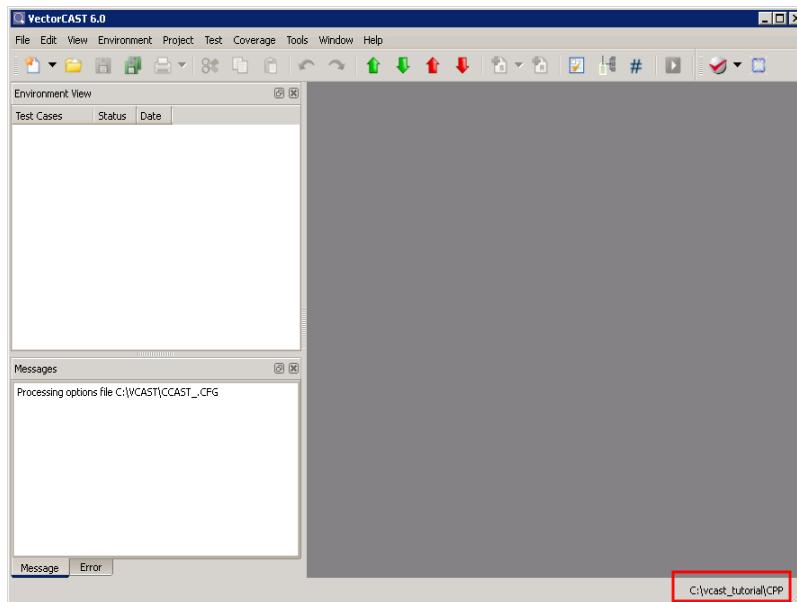
The working directory contains the files you will use to build your test harness. For the purposes of this tutorial, the working directory is **vcast_tutorial\CPP**

For details on starting VectorCAST, refer to "Introduction" on page 7.

1. With the VectorCAST main window open, select **File => Set Working Directory**.
A browsing dialog appears.
2. Navigate to **C:\vcast_tutorial\CPP** ; click **Choose**.



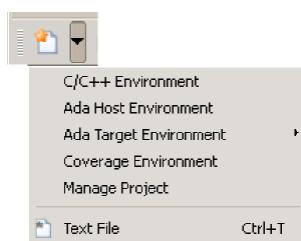
The location and name of the working directory you specified appear at the lower right-hand corner of the VectorCAST main window:



Specifying the type of environment to build

1. Click the **New** button  on the toolbar.

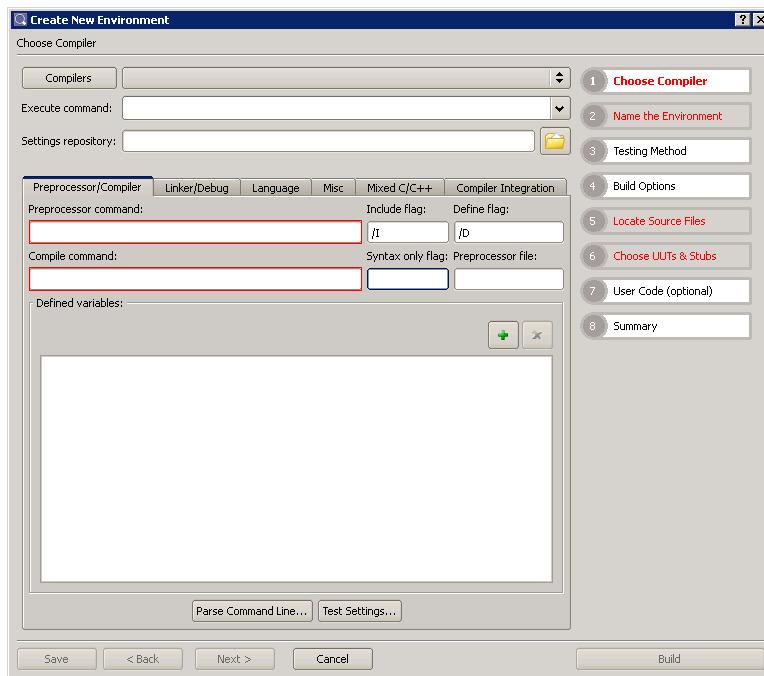
If more than one VectorCAST product is licensed, a drop-down menu appears listing the available environment types:



If the only type of environment licensed is C/C++, the Create New Environment wizard appears when you click the New button.

2. Select **C/C++ Environment**.

The Choose Compiler page of the Create New Environment wizard appears:



Selecting a compiler

You use the Choose Compiler page to specify a compiler. The compiler you specify must be included in your PATH environment variable.

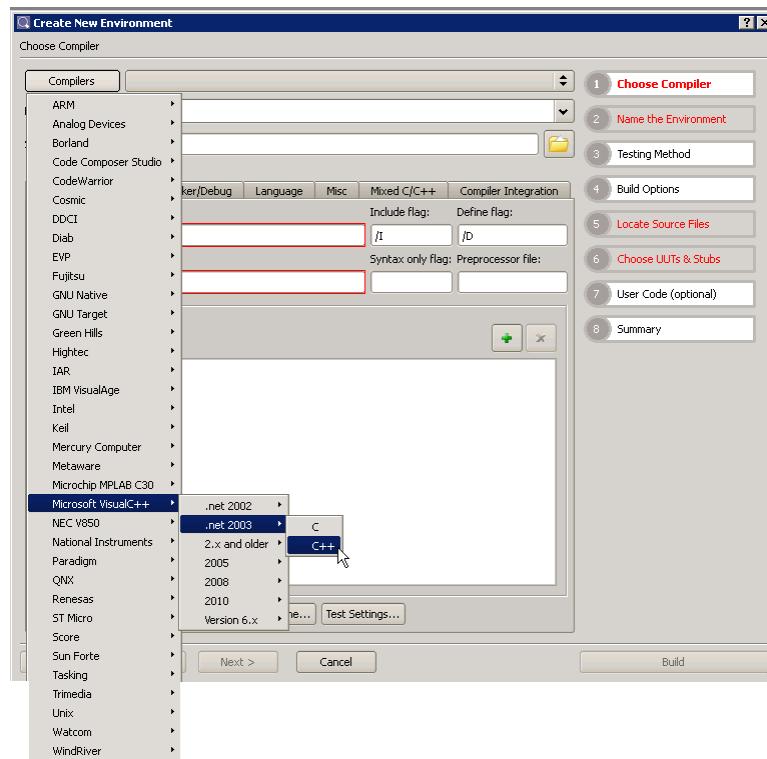
- To access a list of compilers supported by VectorCAST, click the down arrow on the Compilers button. Select a C++ compiler from the dropdown menu.

In this tutorial, you are building a C++ environment; therefore, you must select a C++ VectorCAST compiler template. Most compiler templates have both C and C++ variants.

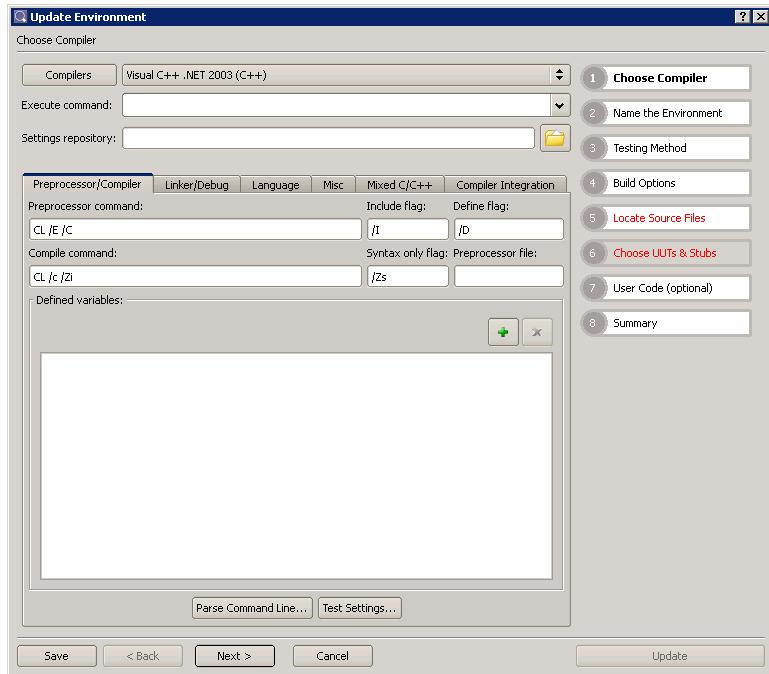


Tip: If you are planning to use VectorCAST with a target compiler, it is recommended you first go through the tutorials using a Host version of your compiler. Once you are familiar with the basic functionality of VectorCAST, refer to the VectorCAST/RSP User's Guide for information specific to your target.

For example, if you are using the Visual C++ .NET 2003 (C++) compiler, you would select from the cascading menus as shown here



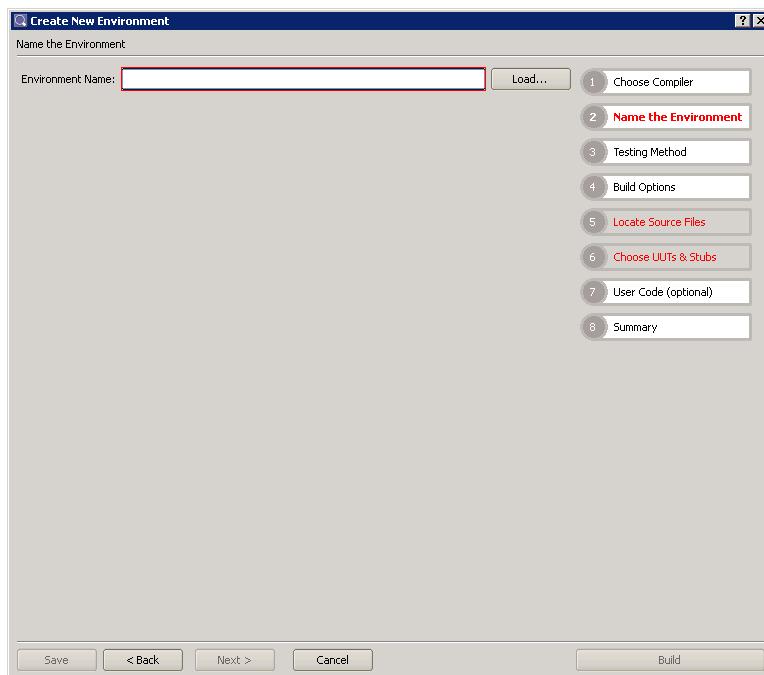
After you select a compiler, VectorCAST fills in the Preprocessor command and Compile command fields with the correct commands.



If VectorCAST cannot find these commands on your default path, it outlines the fields with a red border. In this case, hover your cursor over either field and a tool tip will explain the problem; then click **Cancel**, exit VectorCAST, and address the problem. (Most likely, the compiler you specified is not included in your PATH environment variable.)

2. Click **Next**.

The Name the Environment page appears:

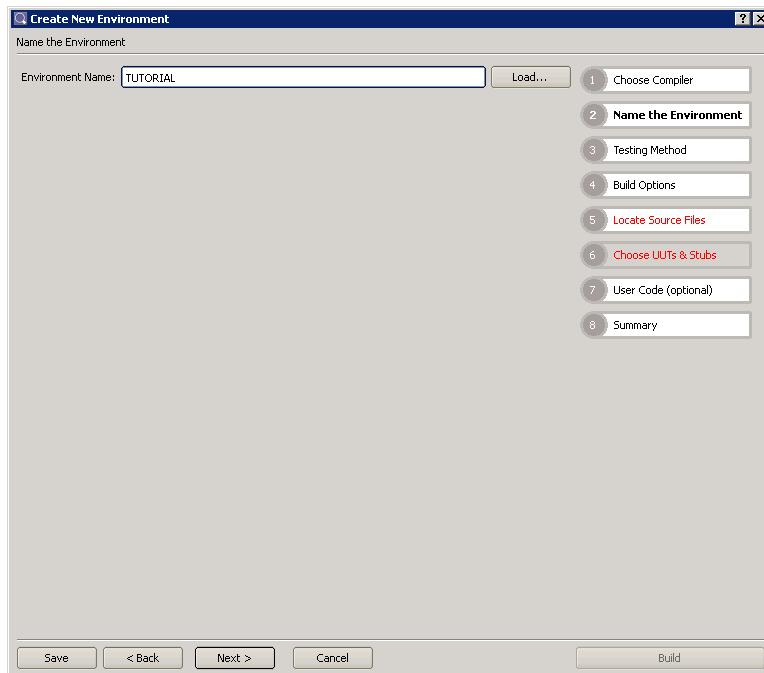


Naming your test environment

For the purposes of this tutorial, you will name your environment 'tutorial'.

1. Enter **tutorial** into the Environment Name field.

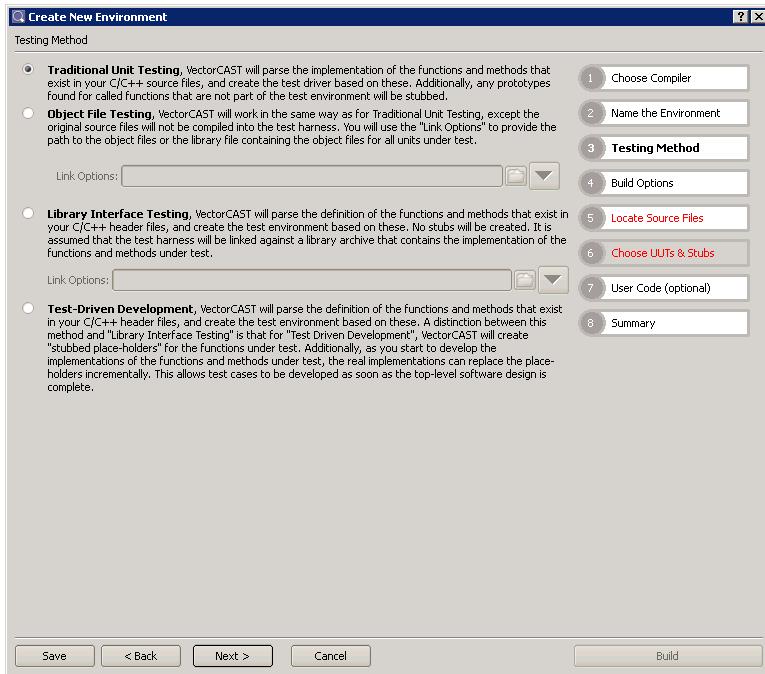
VectorCAST echoes the letters in uppercase:



2. Click **Next**.

Choosing the testing method

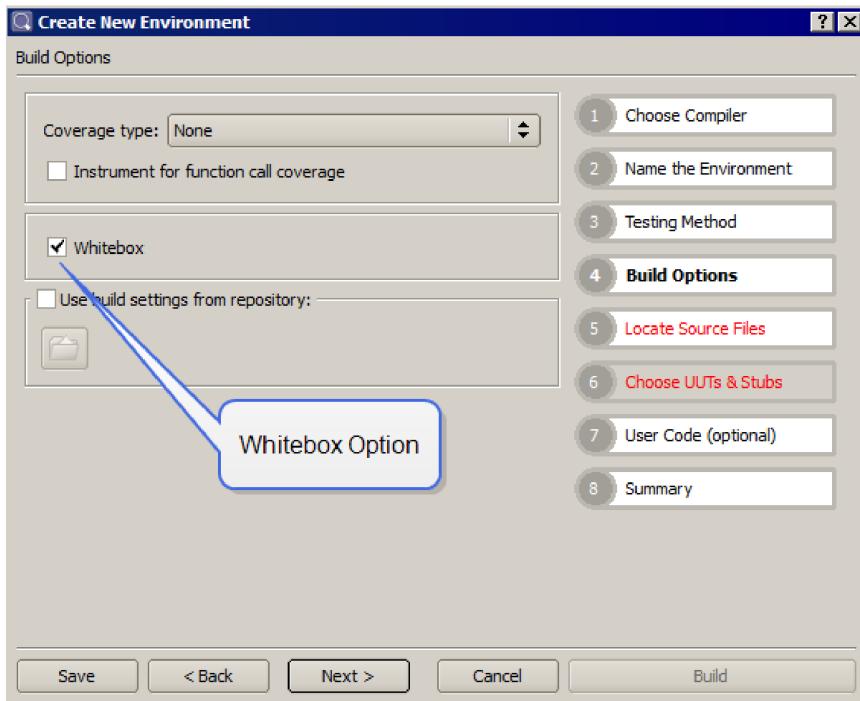
The Testing Method page opens:



This page of the Wizard offers four choices for the method of testing to use. Depending on the testing method, VectorCAST builds a different test harness. For the purposes of this tutorial, we will go with the **Traditional Unit Testing** method.

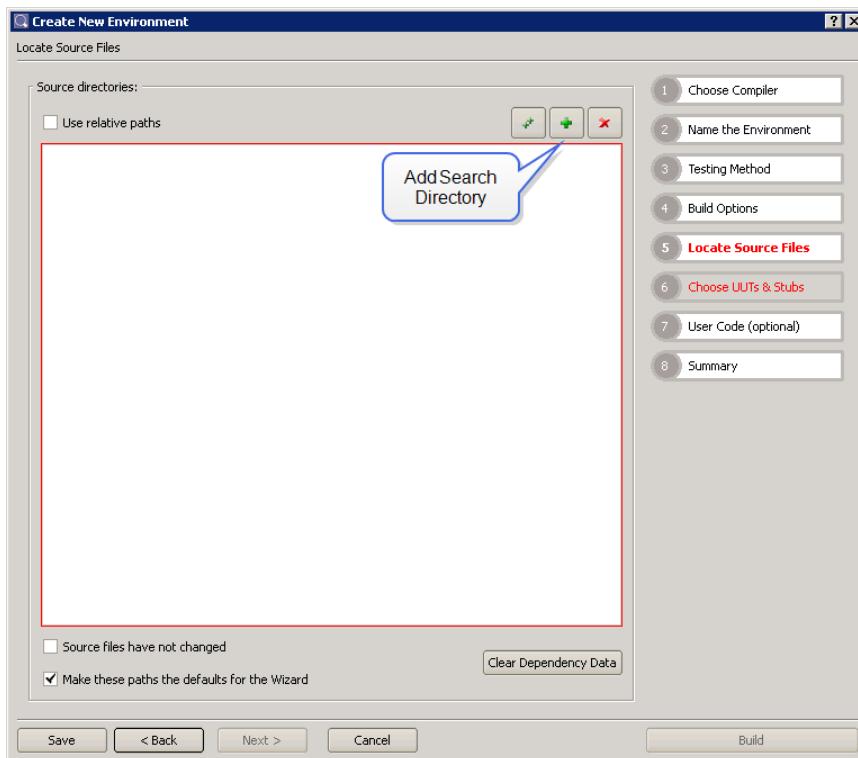
3. Click Next.

The Build Options page opens. Click the checkbox next to “**Whitebox**.” Turning on whitebox ensures you access to private and protected member variables in the source code units.



4. Click **Next**.

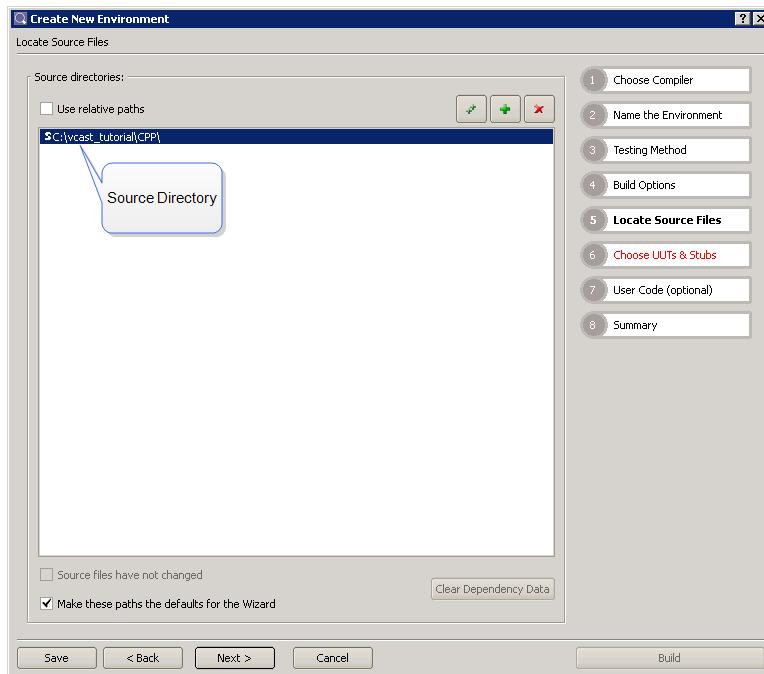
The Locate Source Files page opens:



Specifying the source files to be tested

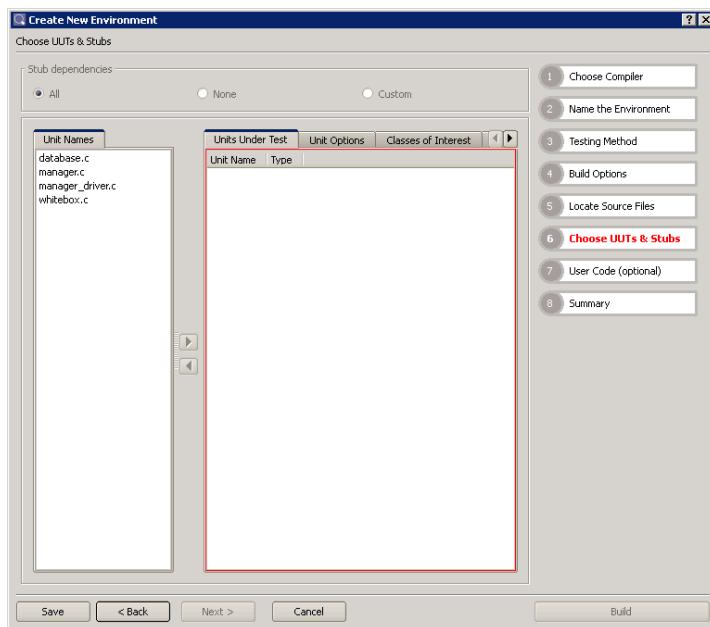
To specify the directory containing the source files to be tested.

1. Click the **Add Search Directory** button  to display the Add Search Directory dialog. The name of the working directory (`C:/vcast_tutorial/CPP/`) displays in the Look in field; the content of this directory displays in the pane below. In this tutorial, you will be building your environment in the same directory in which the source files are located. In practice, you might need to navigate to a different directory.
2. Click **Choose**. The Search directories pane on the Locate Source Files page displays the name of the source directory. The icon  at the left side of the search directory path indicates it is a **Search** directory, so VectorCAST will search it for files that it needs as it is parsing the Unit Under Test.



3. Click **Next >**.

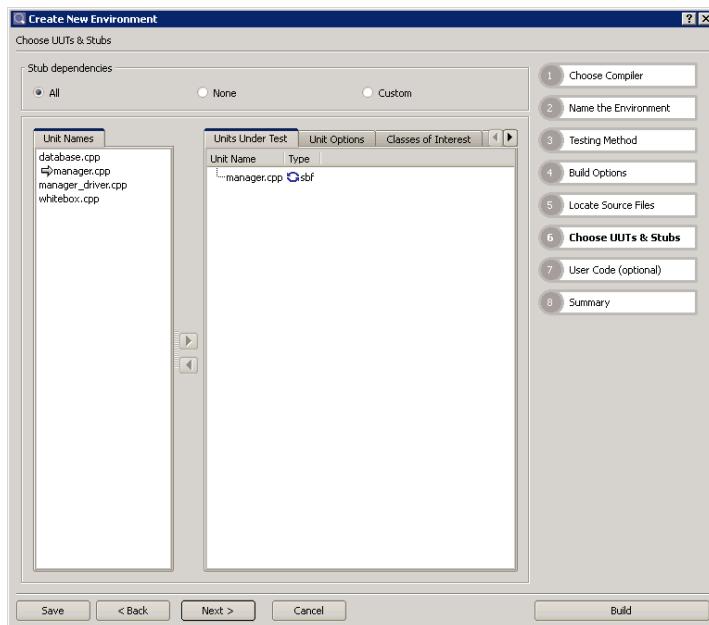
The Choose UUTs & Stubs page appears:

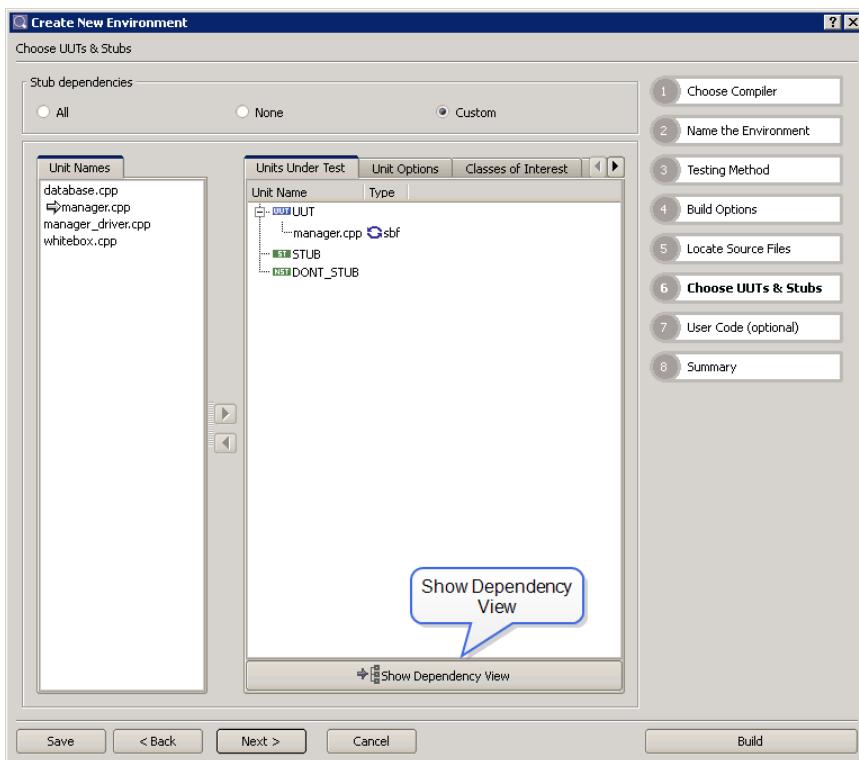


Designating UUTs and stubs

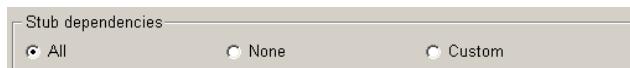
You use the Choose UUTs & Stubs page to designate units as a UUT or stub and the method of stubbing.

The pane under Unit Names lists the units that VectorCAST found in the specified source directory. You can select one or more of these units to be UUTs. For this tutorial, you will select only manager.





1. Click `manager.cpp` under Unit Names, and then click the move-right button; or simply double-click `manager.cpp`. The name `manager.cpp` moves to the Units Under Test tab, and is displayed with the SBF icon (`sbf`) next to it. By default, the unit is a UUT with Stub-by-function enabled.
2. Three options are available under Stub dependencies for designating stubbing: All, None, or Custom:

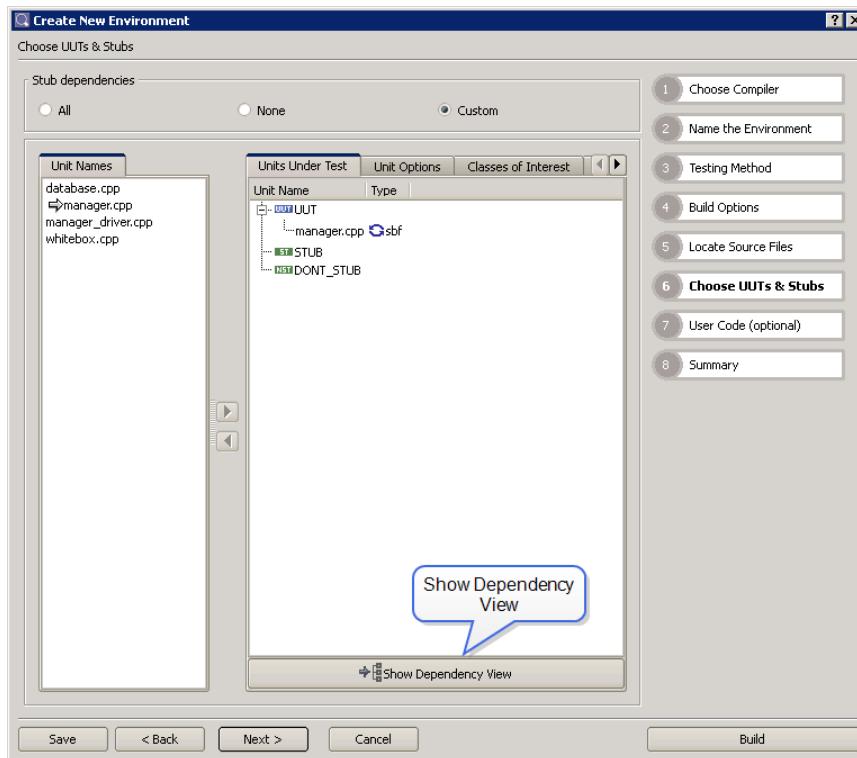


As indicated, VectorCAST defaults to stubbing all dependent units. You can override this setting by selecting None or Custom. The Custom option allows you to select individual units for stubbing by implementation or not stubbing.



Note: By default, VectorCAST performs stubbing By Prototype.

Steps 2 – 4 are optional.



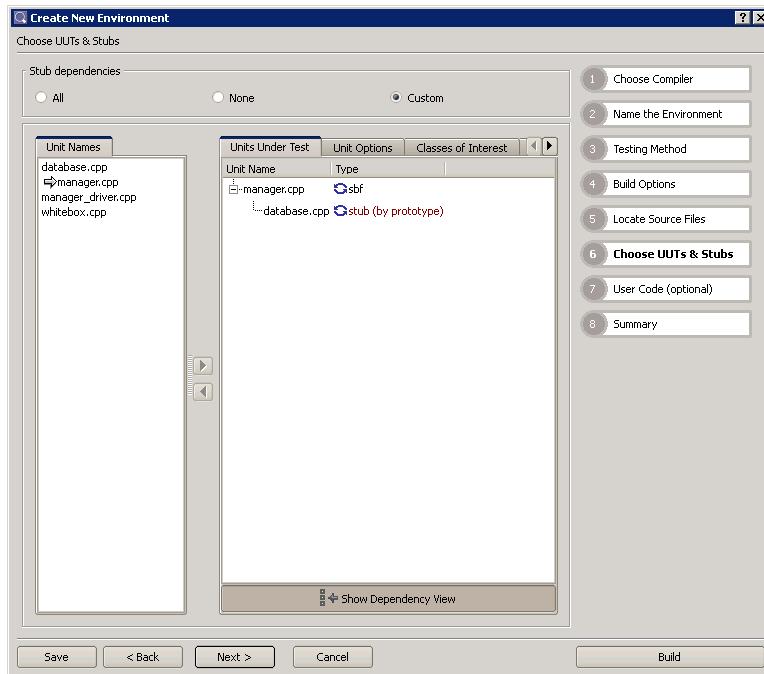
2. Click the **Custom** radio button.

A toggle button (Show Dependency View) appears at the bottom of the Units Under Test tab:

3. Click the **Show Dependency View** button.

VectorCAST parses the units in the search directory, and finds that the unit database is a dependent of the unit manager. It draws the hierarchy to represent this relationship between the units.

Database appears as a dependent to manager, and “stub (by prototype)” appears in the Type column for database, to indicate that it is going to be stubbed using its prototype:

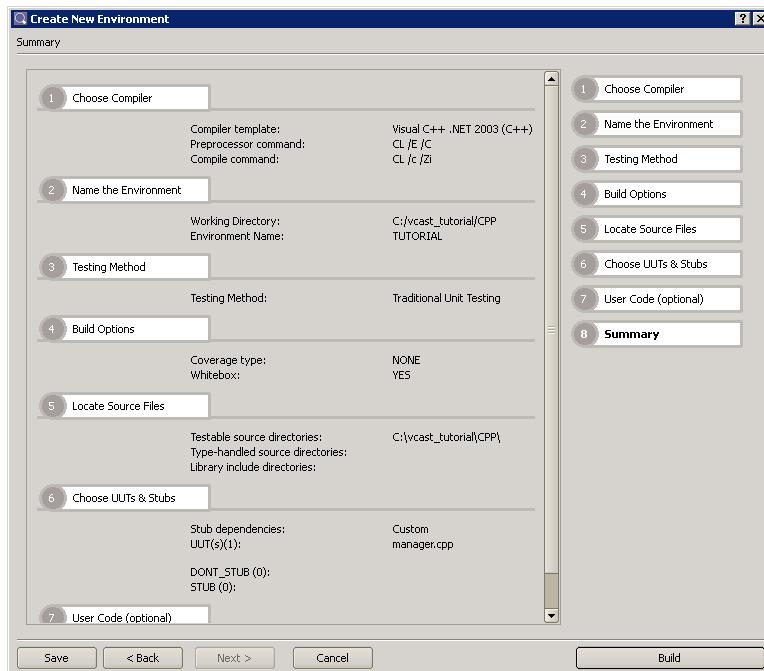


4. Click the cycle icon  for the unit database.
Rotate through the three options: stub (by implementation), not stubbed, stub (by prototype).
Select the default option: stub (by prototype).
5. Click **Next**.
6. The User Code page appears. The User Code page is optional and not needed for this tutorial.

Viewing a summary of your test-environment specifications

1. Click **Next**.

The Summary page appears:



This page allows you to check your test-environment specifications before you generate the test harness.

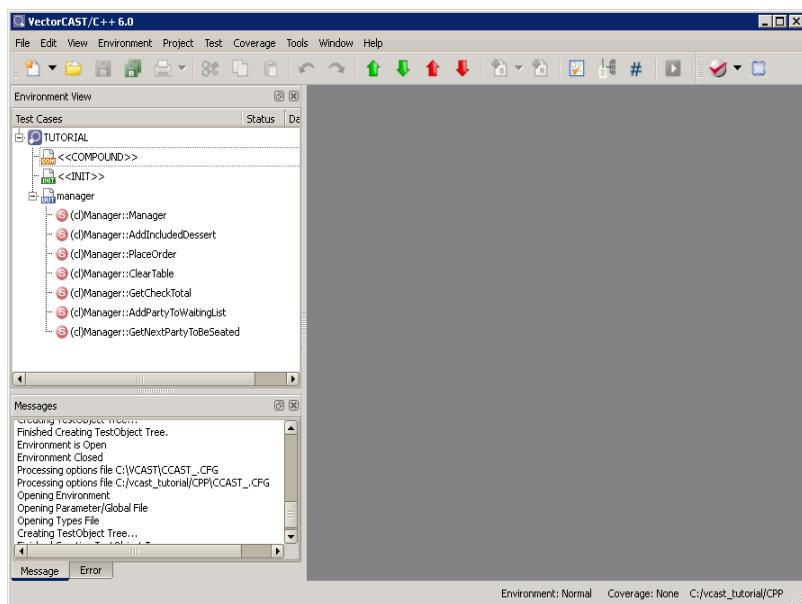
Generating an executable test harness

To compile your specifications into an executable test harness:

1. Click the **Build** button.

As an environment is built, VectorCAST displays a series of messages in the Message window. VectorCAST is parsing `manager.cpp`, creating a stub for `database.cpp`, and compiling and linking the test harness.

When processing has completed, VectorCAST displays the main window:



VectorCAST has created an environment file named `TUTORIAL.vce` and a subdirectory named `TUTORIAL`.

The Environment View shows that the current environment (TUTORIAL) has one UUT, named `manager`, which has six subprograms: `AddIncludedDessert`, `PlaceOrder`, `ClearTable`, `GetCheckTotal`, `AddPartyToWaitingList` and `GetNextPartyToBeSeated`. They are all in the class `Manager`.

At this point, VectorCAST has built the complete test harness necessary to test the `manager` unit, including:

- > A test directory with all test-harness components compiled and linked.
- > The test driver source code that can invoke all subprograms in the file (`manager`).
- > The stub source code for the dependent unit (`database`).

Note what was accomplished during environment creation:

- > The unit under test was parsed. The dependent unit did not have to be parsed, because its prototype was used to determine the parameter profile.
- > The data-driven driver and stub source code were generated.
- > The test code generated by VectorCAST was compiled and linked with the UUT to form an executable test harness.

The harness that VectorCAST builds is *data driven*. This means that data sets can be built to stimulate the different logic paths of the unit under test without ever having to re-compile the harness. You are now ready to build test cases for the subprograms in the unit under test (`manager`).

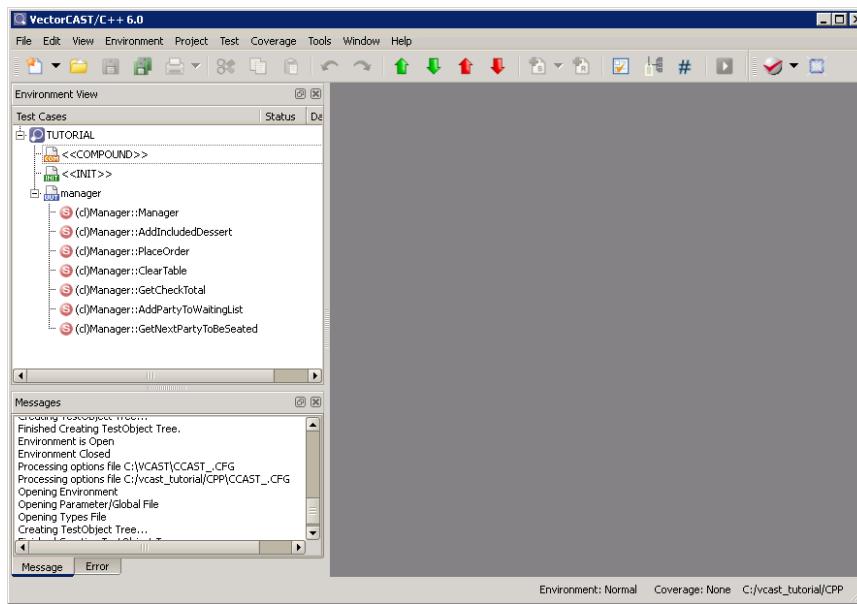
Building Test Cases

In this section, you will build a test case to verify the operation of the subprogram `PlaceOrder` in class `Manager`.

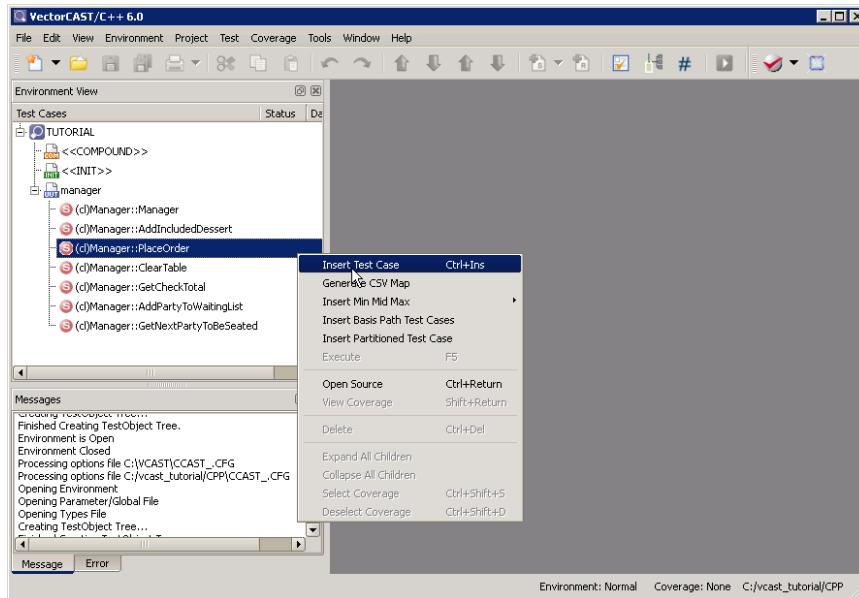
The process of building test cases with VectorCAST is very simple. All the information is available by means of a point-and-click interface. The idea is to create a set of data that will stimulate the unit under test and cause it to execute a particular logic path. In its simplest form, a test case consists of values for the formal parameters of a single subprogram within the unit under test. A more complex test case can also contain values to be returned by a stubbed dependent, or values to be set into global data objects.

Test cases are constructed to stimulate individual subprograms. Each subprogram within a unit will have its own series of test cases defined. The first step is to select a subprogram in the Environment View and create a test case for it.

1. If a list of subprograms is not displayed under `manager` in the Environment View, click the symbol preceding its name to display the list of subprograms.
- The prefix "(cl)Manager" indicates a member function of class Manager.



2. Right-click `PlaceOrder` and select **Insert Test Case** from the popup menu:



In the Environment View, (CL)MANAGER::PLACEORDER.001 appears as a test case under Place_Order:

USER_GLOBALS_VCAST	A collection of global objects for use as a workspace
manager	The unit under test
<<SBF>>	A collapsed tree of subprograms (other than the subprogram under test) which are available to be stubbed in this test case
<<GLOBAL>>	A collapsed tree of variables of global scope to the unit under test
(cl)Manager::PlaceOrder	The subprogram to be tested
Table	Subprogram parameter of type unsigned short
Seat	Subprogram parameter of type unsigned short
Order	Subprogram parameter of type struct
Stubbed Subprograms	A collapsed tree of stubbed subprograms from dependent unit(s)



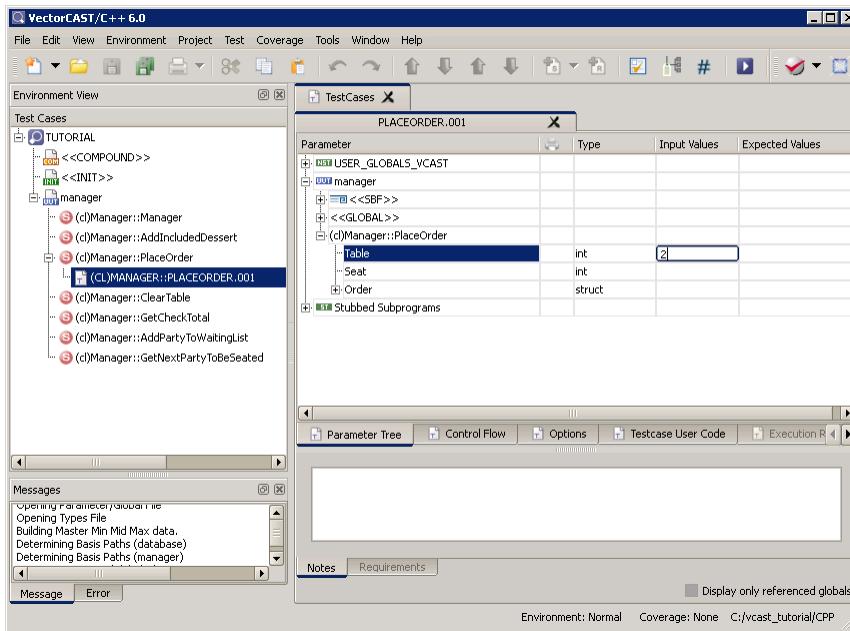
Note: You can change the default name of a test case by right-clicking its name and selecting **Rename** from the popup menu.

The following test case parameter tree appears in the Test Case Editor:

A tree item having branches is preceded by a plus symbol \oplus . To expand a tree item, simply

click the symbol.

- For the parameter **Table**, enter **2** into **Input Values** field:

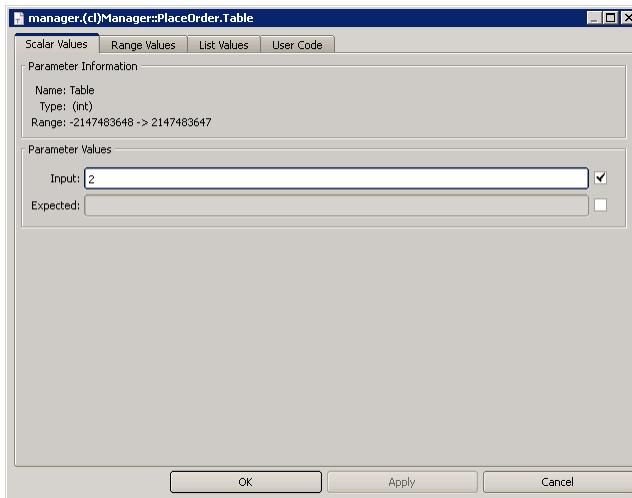


- Press **Enter**.

Table 2 is now the focus of your test case.

- Double-click **Table**.

The parameter dialog box appears:



Notice that you can use this dialog to set simple scalar values, and also to build test cases for a range of values or a list of values, or with user code.

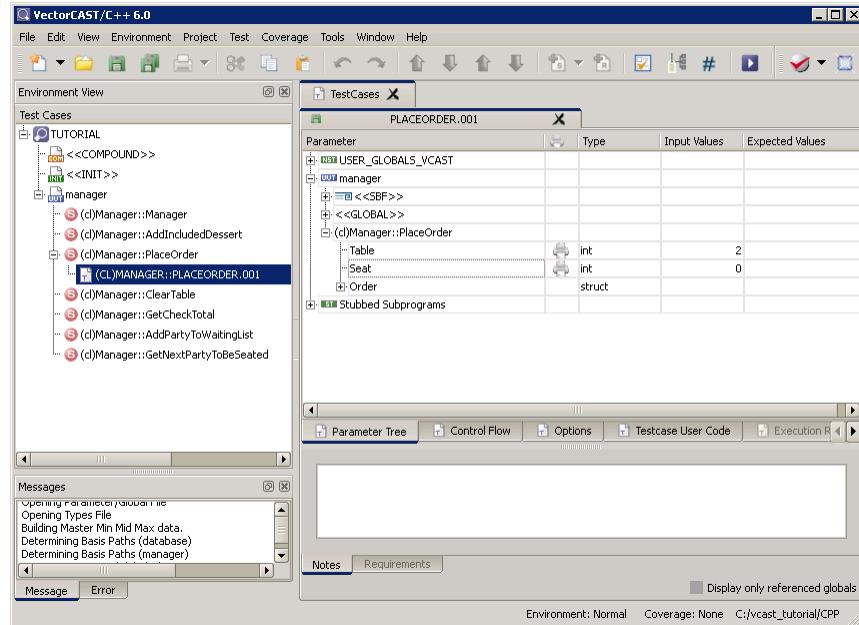
- Click **Cancel** to dismiss this dialog.

You will use it in the next tutorial.



Note: When you create a test case, the only subprogram displayed for the UUT is the subprogram that this test case has been created for, in this case, PlaceOrder.

- For this tutorial, assume now that someone is occupying seat 0 at table 2. Enter **0** into the **Input Values** field for **Seat**, and then press **Enter**.



You will use this same technique to set values for all parameters. Although you can access all parameters of visible subprograms of the unit under test, and of any stubbed unit, it is only necessary to select those parameters that are relevant to the test you are running.

Non-scalar parameters (arrays and structures) are accessed using the same tree-expansion technique that is used for units and subprograms. Parameter **Order** is a structure with five fields. Each field is an enumeration type.

At this point, you could create separate test cases for the other seats at table 2 (as well as for the other tables in the restaurant); however, for the purposes of this basic tutorial, one test case (one order at one seat) is sufficient.

You have someone sitting in seat 0 at table 2; it's time now to take an order.

- Expand **Order** into its fields by clicking .

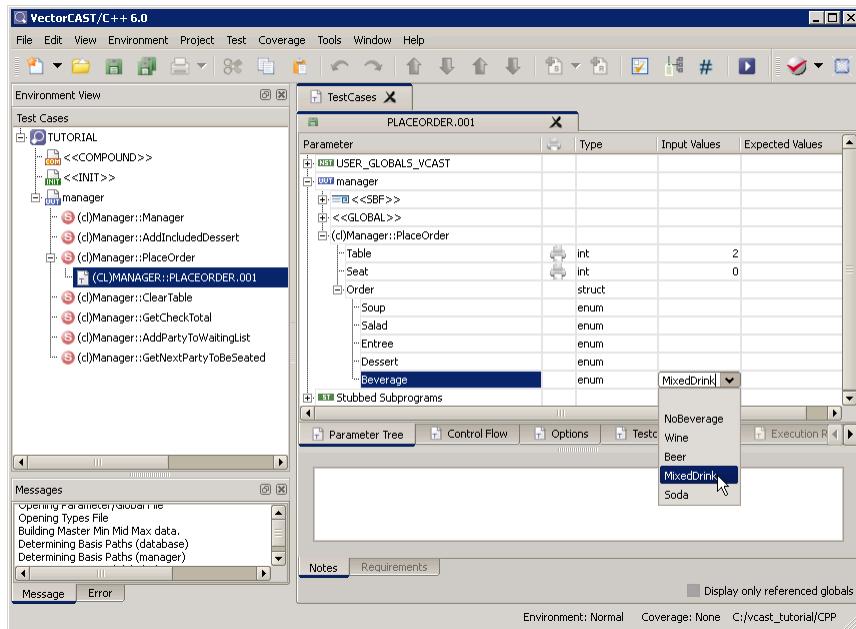
Five order fields appear: Soup, Salad, Entree, Dessert, and Beverage. (The specifications for these order fields are included in "Tutorial Source Code Listings" on page 311.)

- For each order field, click in the **Input Values** column and select a value from the drop-down menu that appears.

For the purposes of this example, select the following values:

Soup **ONION**

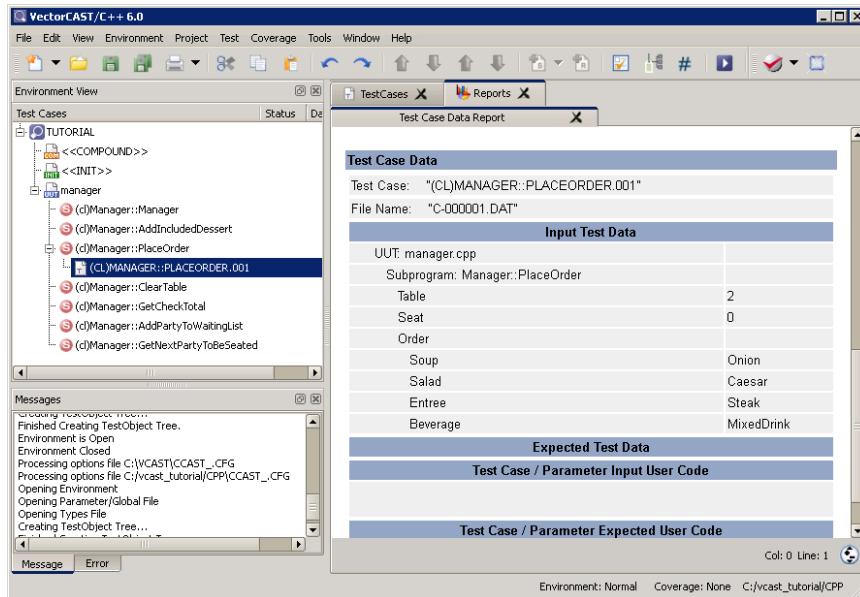
Salad	CAESAR
Entree	STEAK
Dessert	(leave unspecified)
Beverage	MIXED_DRINK



Note: To clear a field, select the blank item at the top of the associated drop-down menu.

10. Save test case PLACEORDER.001: Either select **File => Save** from the main-menu bar or click the Save  button on the toolbar.
11. To access a test-case data listing, select **Test => View => Test Case Data**. The Test Case Data Report for your test case appears in the Report Viewer.
12. In the Table of Contents section of this report, click **Test Case Data**.

The report jumps to the Test Case Data section.



13. Close the report by clicking the X on the report's tab.

You have now built a test case that is ready to be executed by VectorCAST, without any compiling of data or writing of test code. The data that was set up for this case corresponds to the formal parameter list for subprogram **PlaceOrder**. You set data for scalar parameters **Table** and **Seat** and for structure parameter **Order**.

At this point you could execute your test case as is, or you could add expected results for comparing against actual results. Because the latter course is generally the more valuable, you will now add expected results to your test case.

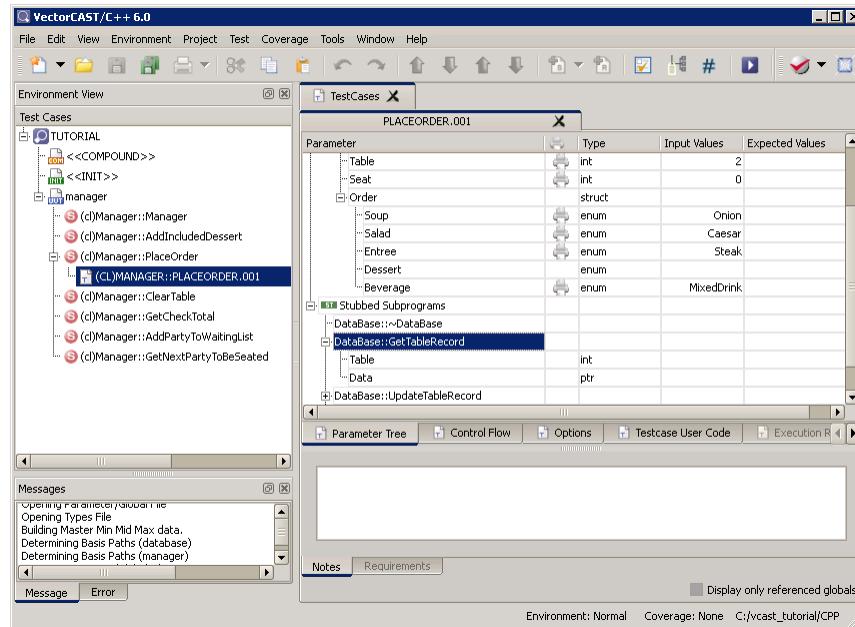
Controlling Values Returned from Stubs

VectorCAST provides you with a simple way to define expected results for each test case. You can add expected results at the time you set input values or at a later time. As a test case is executed, the expected results are compared against actual results to determine whether the test case passed or failed.

In this section, you will add expected results to the test case you created in the previous section:

1. Click the preceding **Stubbed Subprograms**; then click the preceding DataBase::GetTableRecord.

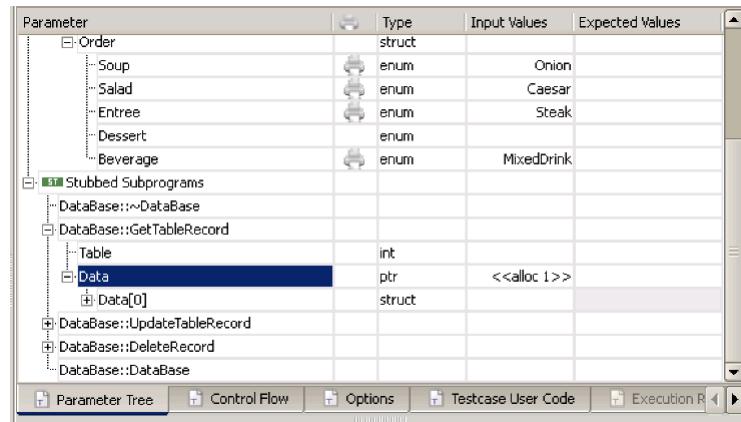
Database subprogram **GetTableRecord** expands into its parameter tree:



If the Database subprograms do not appear under Stubbed Subprograms, see the Troubleshooting section.

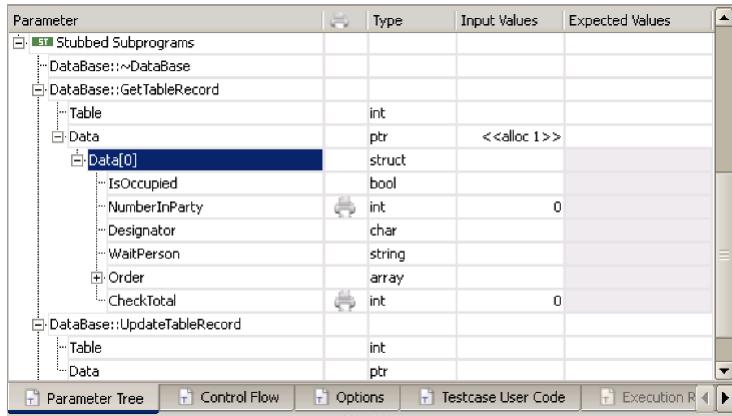
Because Data is a pointer, VectorCAST allows you to allocate a single or multiple instances of the data type that is pointed to.

2. To allocate a single instance, double-click **Data**. An instance named **Data[0]** appears:



3. Click the **+/-** preceding Data[0] to expand it.
4. To initialize the data returned by **DataBase::GetTableRecord**, enter **0** into the respective **Input Values** fields for parameters **NumberInParty** and **CheckTotal**.

5. Press **Enter**.



Parameter	Type	Input Values	Expected Values
Stubbed Subprograms			
DataBase::~DataBase			
DataBase::GetTableRecord			
Table	int		
Data	ptr	<<alloc 1>>	
Data[0]	struct		
IsOccupied	bool		
NumberInParty	int	0	
Designator	char		
WaitPerson	string		
Order	array		
CheckTotal	int	0	
DataBase::UpdateTableRecord			
Table	int		
Data	ptr		

You are now ready to enter expected values for comparing against the actual values returned by `UpdateTableRecord`.



Tip: You might want to make more room by closing up the Parameter trees for both `GetTableRecord` and `manager`.

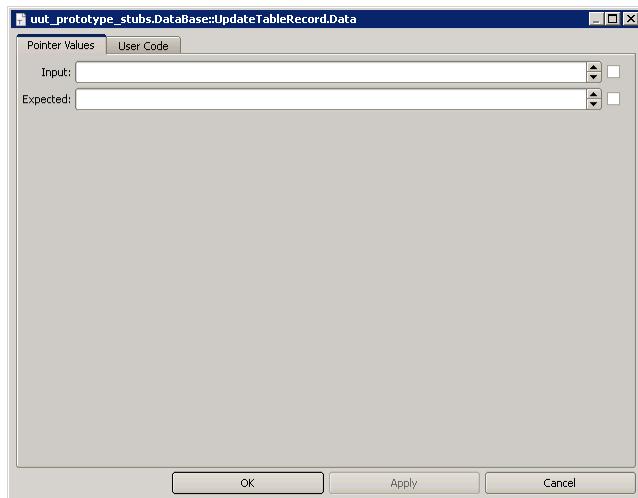
Adding Expected Results to the Test Case

VectorCAST provides you with a simple way to define expected results for each test case. You can add expected results at the time you set input values or at a later time. As a test case is executed, the expected results are compared against actual results to determine whether the test case passed or failed.

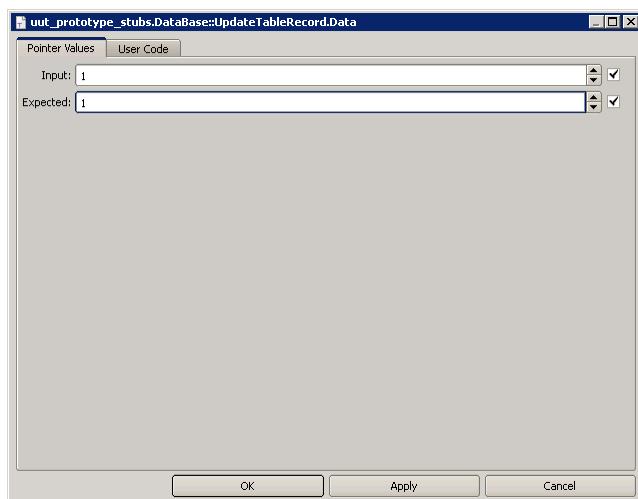
In this section, you will add expected results to the test case you created in the previous section:

1. Click the  preceding **DataBase::UpdateTableRecord**.
`DataBase::UpdateTableRecord` expands into its parameters.
2. To allocate a single instance of the structure for both Input and Expected Values, right-click Data, and choose Properties.

The Parameter dialog opens.



Enable the checkboxes for both Input and Expected, and enter 1 for both.



Click **OK**.

A single element named Data[0] appears in the parameter tree:

Parameter	Type	Input Values	Expected Values
Table			
Data	ptr	<<alloc 1>>	
Data[0]	struct		
IsOccupied	bool		
NumberInParty	int	0	
Designator	char		
WaitPerson	string		
Order	array		
CheckTotal	int	0	
DataBase::UpdateTableRecord			
Table			
Data	ptr	<<alloc 1>>	<<access 1>>
Data[0]	struct		
DataBase::DeleteRecord			
DataBase:: DataBase			

3. Click the preceding Data [0] to expand it.
4. In the Expected Values column, enter the following expected results:

IsOccupied: **true**

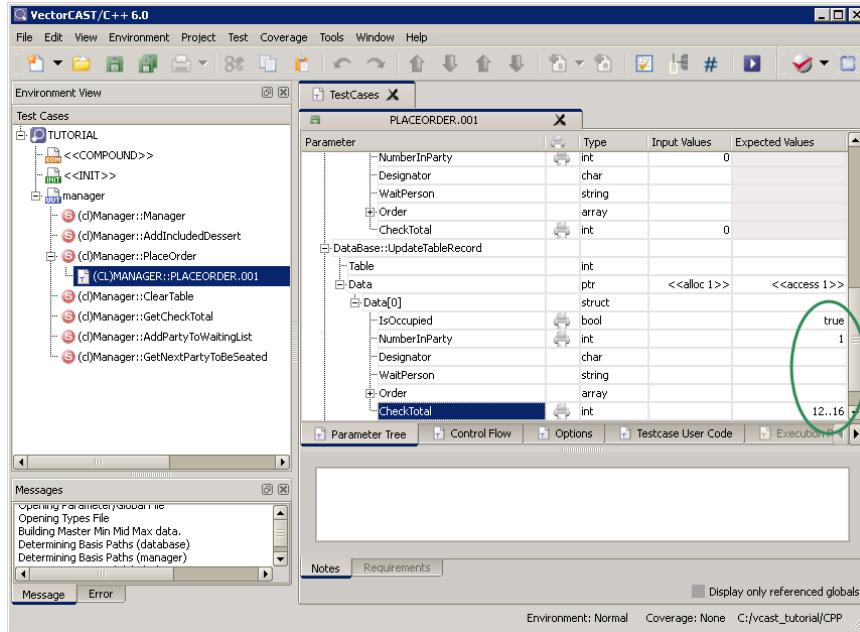
NumberInParty: **1**

Parameter	Type	Input Values	Expected Values
NumberInParty	int	0	
Designator	char		
WaitPerson	string		
Order	array		
CheckTotal	int	0	
DataBase::UpdateTableRecord			
Table			
Data	ptr	<<alloc 1>>	<<access 1>>
Data[0]	struct		
IsOccupied	bool	true	
NumberInParty	int		false
Designator	char		
WaitPerson	string		
Order	array		
CheckTotal	int		



Note: Be sure you enter these values under **Expected Values** column, not into the **Input Values** column.

5. In addition, enter **12..16** as the range of expected results for **CheckTotal**.



6. Expand **Order**.

Order is an array whose index corresponds to the seat specified in the call to **PlaceOrder**. Because we specified seat 0, we need to expand **Order [0]**.

7. In the Input Values column, enter 0 next to <<Expand Indices: Size 4>>.

DataBase::UpdateTableRecord				
Table	int			
Data	ptr	<<alloc 1>>	<<access 1>>	
Data[0]	struct			
IsOccupied	bool		true	
NumberInParty	int		1	
Designator	char			
WaitPerson	string			
+ Order	array	<<Expand Indices: Size 4>>	0	12..16
CheckTotal	int			
DataBase::DeleteRecord				

8. Press **Enter**.

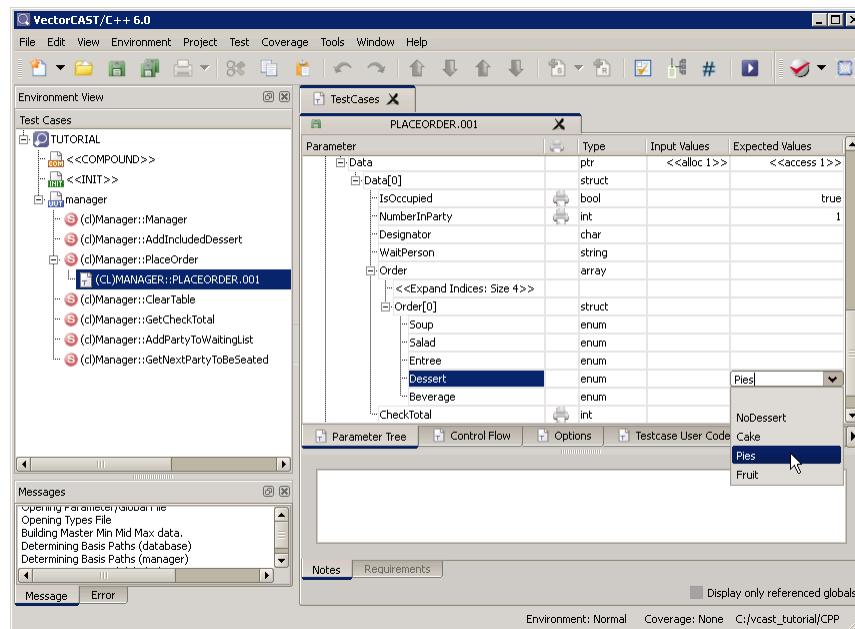
DataBase::UpdateTableRecord				
Table	int			
Data	ptr	<<alloc 1>>	<<access 1>>	
Data[0]	struct			
IsOccupied	bool		true	
NumberInParty	int		1	
Designator	char			
WaitPerson	string			
+ Order	array	<<Expand Indices: Size 4>>		
+ Order[0]	struct			
CheckTotal	int		12..16	

The array element [0] in the array Order is expanded in the Parameter Tree, and the "0" is removed.

9. Expand the tree for Order[0].

	ptr	<<alloc 1>>	<<access 1>>
Data[0]	struct		
IsOccupied	bool		true
NumberInParty	int		1
Designator	char		
WaitPerson	string		
Order	array		
<<Expand Indices: Size 4>>			
Order[0]	struct		
Soup	enum		
Salad	enum		
Entree	enum		
Dessert	enum		
Beverage	enum		
CheckTotal	int		12..16

10. For the Expected Value for Dessert, enter Pies. We expect Pie for dessert because PlaceOrder calls AddIncludedDessert, which sets Dessert to Pie.



You now have a test case defined with input data and expected results.

The values entered into the Expected Values column for a stub are verified as they are passed into the stub; the values entered into the Input Values column are passed out of the stub and back into the test harness.

You must now save your data in order to make them a part of your test case.

When a test case has been modified and needs to be saved, a green icon appears to the left of its name on a tab along the top of the Test Case Editor:

11. To save your modifications, click the **Save button on the Toolbar**.



12. To view the saved data, select **Test => View => Test Case data**, then click the link “Test Case Data.”

Table 1. Test Case Data

Input Test Data	
UUT:	manager.cpp
Subprogram:	Manager::PlaceOrder
Table	2
Seat	0
Order	
Soup	Onion
Salad	Caesar
Entree	Steak
Beverage	MixedDrink
Stubbed Subprograms:	
Unit:	database.h
Subprogram:	DataBase::GetTableRecord
Data	
[0]	
NumberInParty	0
CheckTotal	0
Expected Test Data	
Stubbed Subprograms:	
Unit:	database.h
Subprogram:	DataBase::UpdateTableRecord
Data	
[0]	
IsOccupied	true
NumberInParty	1
Order	
[0]	
Dessert	Pie
Check_Total	BETWEEN:12 AND:16
Test Case / Parameter Input User Code	
Test Case / Parameter Expected User Code	



If your test case Data report does not show any data in the section labeled Expected Test Data, refer to the section titled Troubleshooting at the end of this tutorial.

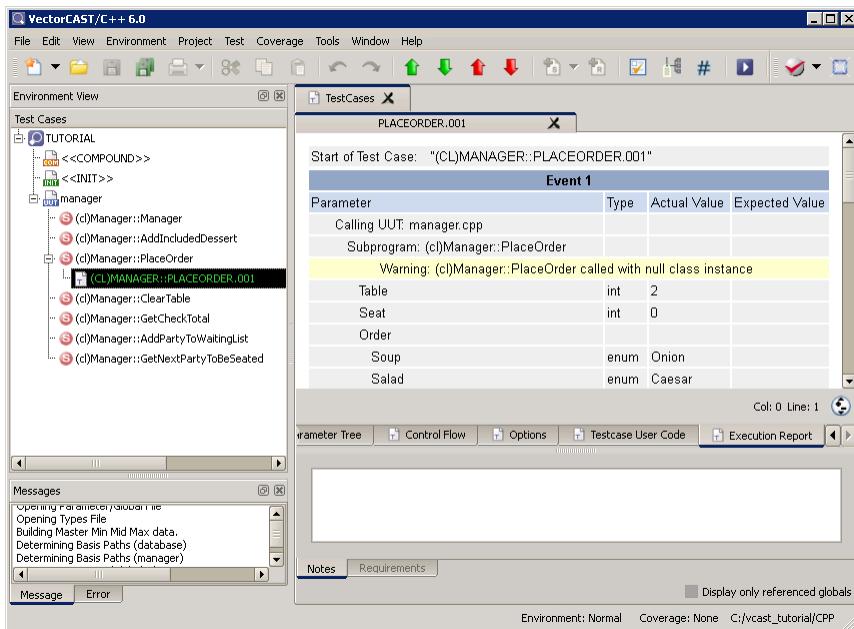
13. To close this report, click the X on the Test Case Data tab. To close all reports, click on the X on the Reports tab.

You have created a test case named PLACEORDER.001 that includes expected results. You are now ready to execute it.

Executing Test Cases

In this section, you will execute the test case you created in the previous sections.

1. To execute your test case, click **(CL)MANAGER::PLACEORDER.001** in the Environment View, then click the **Execute** button  on the toolbar.
When processing completes, an execution report appears in the Test Case Editor

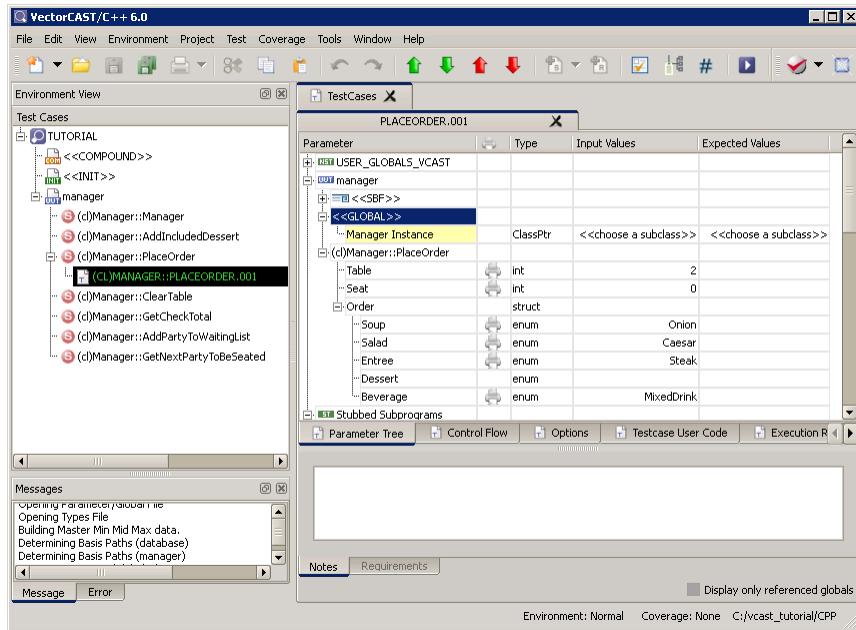


Instantiating a Class Instance

Because the test case did not specify a class instance, a reminder appears in the Execution Report: Warning: (cl)Manager::PlaceOrder called with null class instance

Because manager.cpp is a simple tutorial source file, this warning is not a problem. However, in most situations, member data would be accessed during the test execution, and so it is best to instantiate a class instance.

1. Return to the test's Parameter Tree by clicking the Parameter Tree tab along the bottom of the Test Case Editor.
2. Expand the UUT, manager, then expand <<GLOBAL>>.



In a typical source code unit, there may be many class instances listed. To determine the class instance that is associated with the subprogram under test, look for the yellow-shaded one.

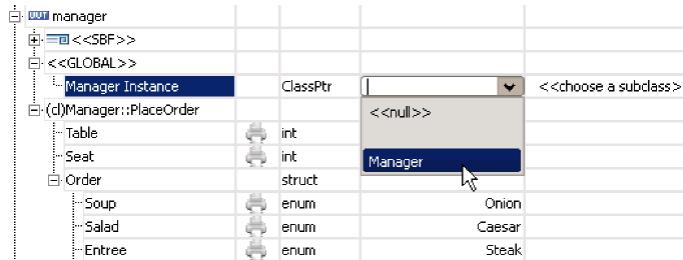
3. Next to the yellow-shaded class instance, Manager Instance, click the Input Values column cell containing <<choose a subclass>>, and select **Manager** from the drop-down list.



Note: Typical source code would have many subclasses to choose from in this drop-down list.

4. Press Enter.

The subclass Manager is displayed.



When the source code defines only one constructor, it is selected by default. Under ordinary circumstances, you would need to click the Input Values cell containing <<choose a constructor>>, and select a constructor from the drop-down list.



Note: Typical source code would have many constructors to choose from in this drop-



down list.

The collapsed member variables are expanded, public as well as private because this environment is a whitebox environment. For the purposes of this tutorial, it is not necessary to initialize member variables here.

	ClassPtr	Manager	<<choose a subclass>>
	SubClass	Manager::Manager()	
Manager Instance			
Manager			
Manager::Manager			
MemberVariable	int		
Data	class		
WaitingList	array		
WaitingListSize	unsigned int		
WaitingListIndex	unsigned int		
(c)Manager::PlaceOrder			

5. Save the test case by clicking the Save icon on the toolbar .

6. Execute the test case again by clicking the Execute icon on the toolbar .

Click the green down-arrow  on the toolbar, or scroll down the report until you come to the green-shaded results section.

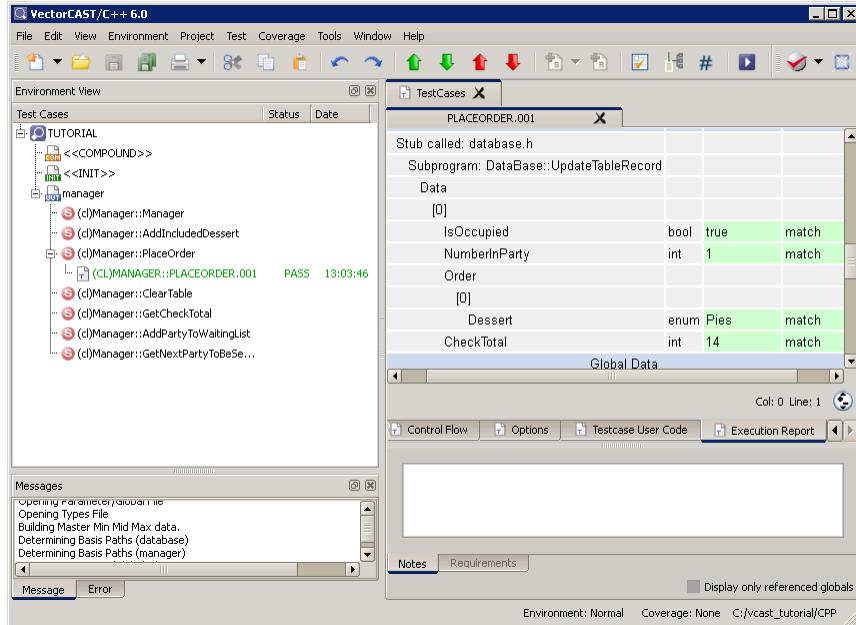
The green shading and the term “match” here indicate that the expected values you specified were matched by actual values. Your test passed!

Parameter	Type	Expected	Actual	Status
Data	[0]			match
IsOccupied	bool	true		match
NumberInParty	int	1		match
Order	[0]			
Dessert	enum	Pies		match
CheckTotal	int	14		match

- 7.

Note that “(CL)MANAGER::PLACEORDER.001” in the Environment View is now green. This color-coding signifies that the most recent execution of test case PLACEORDER.001

passed. In addition, the Status field displays PASS in green, while the Date field displays the hour:minute:second of the execution.



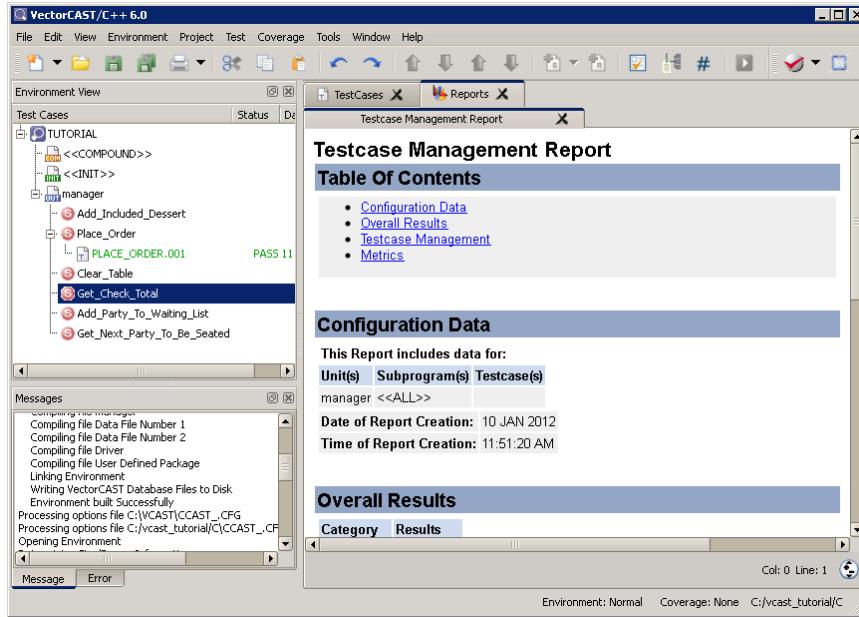
Generating Test Reports

In this section, you will view the Test Case Management Report for the recent execution of test case PLACEORDER.001.

A Test Case Management Report includes:

- > The results of all tests conducted with this environment
 - > The number of test cases that passed
 - > The number of comparisons made that passed
 - > The pass-fail status of all test cases
 - > Metrics data, including cyclomatic complexity and, if code coverage was enabled, the coverage achieved
1. To view the Test Case Management Report for your environment, select **Test => View => Test Case Management Report** from the main-menu bar.

The Test Case Management Report for environment TUTORIAL appears in the Report Viewer:



Note: You can customize the appearance of this or any other VectorCAST report. To customize a report, select **Tools => Options**; click the **Report** tab; then click the **Format** sub-tab.

2. Use the scrollbar to move through the report.
3. To print your Test Case Management Report, click the **Print** button on the toolbar  , or select **File => Print** from the main-menu bar.
4. To save your report into a file, select **File => Save As**.

When the **Save As** dialog box opens, enter **tutorial_report.html** for the file name, then click **Save**.



Note: You can open a saved report with any HTML browser (Internet Explorer, Mozilla, Netscape, etc.).

Analyzing Code Coverage

Code coverage analysis enables you to see which parts of your application are being stimulated by your test cases. Once coverage is initialized for an environment, you can choose to run your VectorCAST test cases with either the coverage-instrumented test harness or the uninstrumented harness.

Coverage enables you to view a code coverage report for individual test cases and an aggregate report showing coverage resulting from any subset of test cases.

The Units Under Test must be instrumented for the kind of coverage you want: Statement, Branch, Statement + Branch, and Statement + MC/DC, and MC/DC. In this section, you will instrument the source files in your environment for statement coverage:

- To instrument the source files in your environment for statement coverage, select **Coverage => Initialize => Statement**.

The status bar on the bottom right of the VectorCAST main window now indicates that statement coverage is active:



- To execute an existing test case and generate a report, click **PLACEORDER.001** in the Environment View, then click the execute button (▶).

When coverage is enabled, VectorCAST keeps track of the lines or branches in the source code that are traversed during execution. You can access a report on the results by way of clicking the checkbox next to the test case name in Environment View.

- To access the coverage report for your test case, click the **green checkbox** to the left of **PLACEORDER.001**.



An annotated version of the `manager` source code appears in the Coverage Viewer:

A screenshot of the VectorCAST Coverage Viewer. The left pane shows the Test Cases view with the test case "(CL)MANAGER::PLACEORDER.001" selected. The right pane displays the annotated source code for the `Manager::PlaceOrder` function. The code is color-coded to show coverage status: green for exercised statements, red for unexecuted statements, and black for continuation or unexecuted statements. An asterisk (*) is placed in the first column of each row to indicate a covered statement. A blue arrow points to the current line (line 2, statement 5). The status bar at the bottom shows "Statements 39%".

```

Statements 39%
Order->Beverage == MixedDrink)
2 4      *
    Order->Dessert = Pies;
}
else
2 5      if(Order->Entree == Lobster &&
    Order->Salad == Green &&
    Order->Beverage == Wine)
{
    Order->Dessert = Cake;
}
void Manager::PlaceOrder(int Table, int Seats)
{
3 1      * TableData Type TableData;
3 2      * Data.DeleteTableRecord(&TableData);
3 3      * Data.GetTableRecord(Table, &TableData);
3 4      * TableData.IsOccupied = true;
3 5      * TableData.NumberInParty++;
3 6      * TableData.Order[Seat] = Order;
3 7      * /* Add a free dessert in some cases */
    AddIncludedDessert(&TableData.Order[Seat]);
3 8      * switch(Order.Entree)
{
}

```

Each set of repeating numbers in the first column marks a subprogram within `manager`. The ascending numbers in the second column mark the executable statements within a subprogram. The statements in green have been exercised; the statements in red have not been exercised; the statements in black cannot be exercised, or are the continuation of previous statements. An asterisk (*) indicates a covered statement. The blue arrow marks the current line. You can move the current line ↗ by means of the up and down arrows on your keyboard, or by clicking on

a line.

Note the icons on the left-hand margin of the Coverage Viewer:

```

void Manager::PlaceOrder(ir
{
    TableDataType TableData;
    Data DeleteTableRecord(&TableData);
    Data GetTableRecord(TableData);
    TableData.IsOccupied = tr;
    TableData.NumberInParty++;
    TableData.Order[Seat] = C;

    /* Add a free dessert in
     * AddIncludedDessert(&TableData);
     */
    switch(Order:Entree) {
        case Steak:
            TableData.CheckTotal();
            break;
        case Chicken:
            TableData.CheckTotal();
    }
}

```

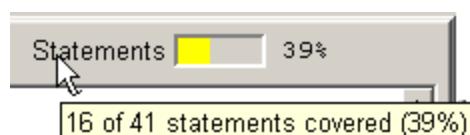
A circled minus-sign means the associated subprogram is fully expanded in the Coverage Viewer, and can be collapsed. To collapse a subprogram (to its first line), double-click either the circle or the first line of the subprogram.

Clicking the file symbol gives you immediate access to the associated source code.

The status bar (upper right) tells you the number and percentage of statements exercised.

4. Hover your mouse over **Statements** on the status bar.

A tool tip provides the exact number of lines covered; in this case, 16 of 41 (39%).



Note: VectorCAST's animation feature allows you to view the step-by-step coverage of a test case. This feature is described in the user guide accompanying your version of VectorCAST.

The Metrics tab at the bottom of the Coverage Viewer displays a tabular summary of the coverage achieved, on a per subprogram basis.

5. Click the **Metrics** tab:

Subprogram Name	Complexity V(g)	Statements
Manager::Manager	1	2 / 2
Manager::AddIncludedDessert	4	3 / 6
Manager::PlaceOrder	5	11 / 18
Manager::ClearTable	1	0 / 1
Manager::GetCheckTotal	1	0 / 4
Manager::AddPartyToWaitingList	3	0 / 7
Manager::GetNextPartyToBeSeated	2	0 / 3
TOTAL	17	16 / 41

This table tells you that subprogram **PlaceOrder** consists of 18 executable statements; that 11 of these statements were exercised during the selected test; and that the complexity of the subprogram is 5, meaning there are five distinct execution paths.

6. Click the **Coverage** tab to return to the previous view.

For each green line, you can identify the test case that most recently exercised it.

7. Hold your mouse over any green line in the Coverage Viewer.

A tool tip appears showing the name of the test case that exercised this line:

```

    void Manager::PlaceOrder(int Table, int Seat, Order* Order)
{
    TableData* TableData;
    Data.DeleteTableRecord(&TableData);
    Data.GetTableRecord(Table, &TableData);
    TableData.IsOccupied = true;
    TableData.NumberInParty++;
    TableData.TableID = [CL]MANAGER::PLACEORDER.001;
    /* Add a free dessert in some case */
    AddIncludedDessert(&TableData.Order[Seat]);
    switch(Order_Entree) {
        case Steak :
            TableData.CheckTotal += 14;
            break;
        case Chicken :
            TableData.CheckTotal += 10;
            break;
        case Lobster :
            TableData.CheckTotal += 18;
    }
}

```



Note: If you had run multiple test cases with coverage enabled, and had multiple check boxes selected in the Environment View, the tool tip would list all cases that provided coverage for the selected line.

8. To access the test case that exercised a line, right-click on the line, then select the test case from the popup menu that appears; in this example, select **PLACEORDER.001**:

```

3 1   *
3 2   *
3 3   *
+ 3 4   *
3 5   *
3 6   *
3 7   *     void Manager::PlaceOrder(int Table, int Seat, Or
3 8   *     { TableDataType TableData;
3 9   *       Data.DeleteTableRecord(&TableData);
3 10  *      Data.GetTableRecord(Table, &TableData);
3 11  *      TableI   Expand Subprograms
3 12  *      TableI   Collapse Subprograms
3 13  *      /* Add
3 14  *      (CL)MANAGER.PLACEORDER.001
3 15  *      AddIn
3 16  *      switch(Order_Entrée) {
3 17  *        case Steak :
3 18  *          TableData.CheckTotal += 14;
3 19  *          break;
3 20  *        case Chicken :
3 21  *          TableData.CheckTotal += 10;
3 22  *          break;
3 23  *        case Lobster :
3 24  *          TableData.CheckTotal += 18;
3 25  *      }
3 26  *    }
3 27  *  }
3 28  *}

```

The test case **PLACEORDER.001** opens in the Test Case Editor.

- To return to the Coverage Viewer, right-click on **manager** in the **Environment View**, then select **View Coverage** from the popup menu.



Note: You could also select **Window => Coverage**.



Note: You could also select **Window => Coverage Viewers** (**Coverage Results for MANAGER**).

Generating and Managing Coverage Reports

In this section, you will view the Aggregate Coverage Report for your test environment.

An Aggregate Coverage Report includes:

- > A source-listing for each UUT, indicating the lines covered
 - > A metrics table giving the complexity and the level of coverage for each subprogram
- To view the Aggregate Coverage Report for your test environment, select **Test => View => Aggregate Coverage Report** from the main-menu bar.

The Report Viewer displays a report showing the total coverage for all test cases executed on manager. The associated source code displayed is color-coded to mark the exercised lines (green) and the un-exercised lines (red):

```

Code Coverage for Unit: manager
-- Coverage Type: statement
-- Unit: manager
-- Test Case: Aggregate
--> void Manager::AddIncludedDessert(OrderType)
{
    if(!Order)
        return;
    if(Order->Entree == Steak &&
        Order->Salad == Caesar &&
        Order->Beverage == MixedDrink)
        WaitingListSize = 0;
    WaitingListIndex = 0;
}

```

2. Scroll down the report until you come to a table similar to the metrics table you accessed from the Metrics tab.

The partially exercised subprograms, **PlaceOrder** and **AddIncludedDessert**, are shown in yellow; the other subprograms, all unexecuted, are shown in red. If 100% coverage for a subprogram is achieved, it shows in green:

Unit	Subprogram	Complexity	Statement Coverage
manager	Manager::Manager	1	100% (2 / 2)
	Manager::AddIncludedDessert	4	50% (3 / 6)
	Manager::PlaceOrder	5	61% (11 / 18)
	Manager::ClearTable	1	0% (0 / 1)
	Manager::GetCheckTotal	1	0% (0 / 4)
	Manager::AddPartyToWaitingList	3	0% (0 / 7)
	Manager::GetNextPartyToBeSeated	2	0% (0 / 3)
TOTALS	7	17	39% (16 / 41)
GRAND TOTALS	7	17	39% (16 / 41)

3. To print the Aggregate Coverage Report, click the **Print** button on the toolbar , or select **File => Print** from the main-menu bar.

4. To save your report into a file, select **File => Save As**.

When the Save As dialog box opens, enter tutorial_coverage.html for the file name, then click the **Save** button .

You can open your saved report with any HTML browser (Internet Explorer, Mozilla, Netscape, etc.).

5. Choose **File => Close All** in preparation for the next section.

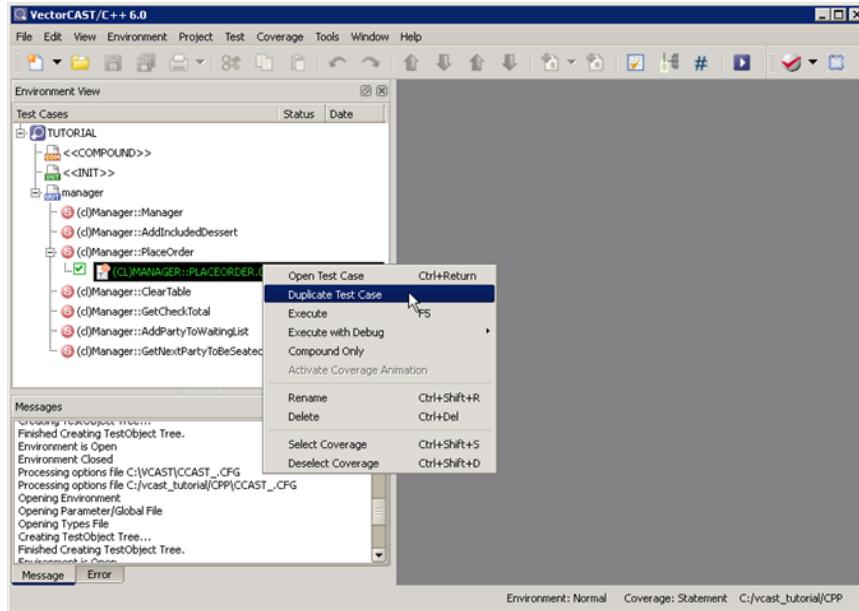
Using Stub-By-Function

In the last section, we tested the subprogram PlaceOrder using inputs of Steak, Caesar, and MixedDrink for the Order. During test execution, PlaceOrder calls the subprogram AddIncludedDessert, which checks for the combination of Steak and Caesar and MixedDrink. It finds them and sets the Dessert to Pie. However, we would like to test for the case when AddIncludedDessert doesn't set the Dessert to Pie.

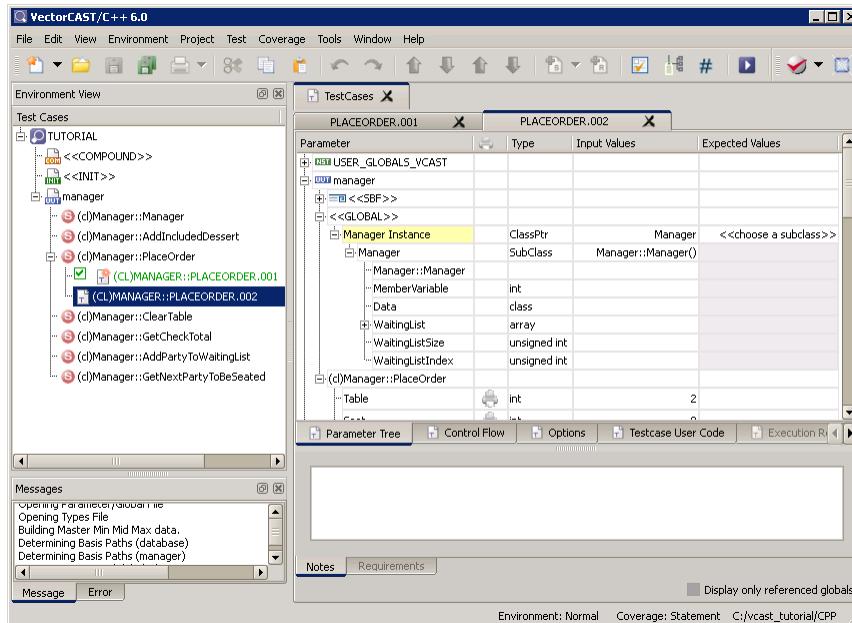
In this section, we will create a new test case which stubs the subprogram AddIncludedDessert, so that we can cause it to return something else for the Dessert, such as Cake.

You will use VectorCAST to:

- > Duplicate a test case
 - > Stub a subprogram in the UUT
 - > Set the value of a parameter in the stubbed subprogram to the desired value, so that it is returned to the test harness instead
 - > Export a test script to see how SBF is specified in a test script
1. In the Environment View, right-click the test case name "PLACEORDER.001" and choose Duplicate Test Case.



A new test case is created, named PLACEORDER.002.

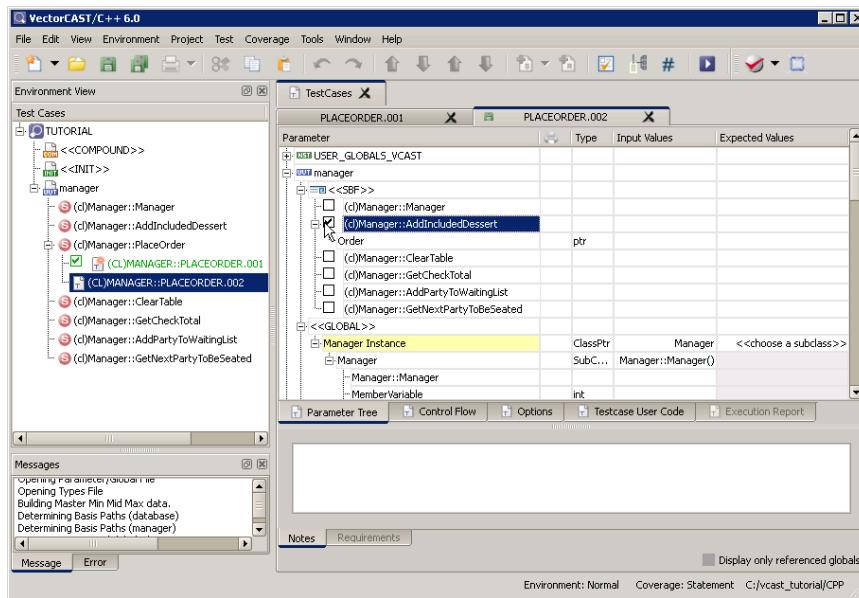


Note: Using drag-and-drop, you can duplicate a test case in another subprogram. Select the test case, press Alt+Shift, drag-and-drop the test case on the destination subprogram. Only data that applies to the parameters in the new subprogram are copied to the new test case.

2. Click preceding <<SBF>>.

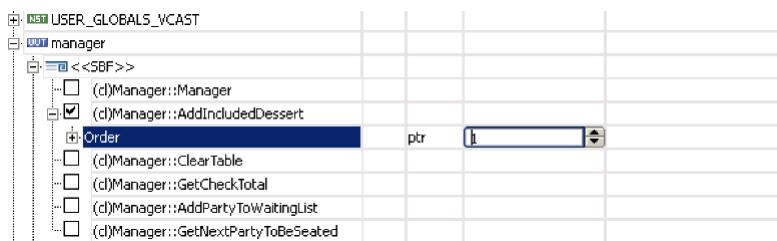
The list of subprograms defined in the UUT is displayed, not including the subprogram under test, **PlaceOrder**. The subprograms in this list are available for stubbing in this test case.

- Click the checkbox next to **AddIncludedDessert**. The checkmark indicates that this subprogram should be stubbed during test execution.



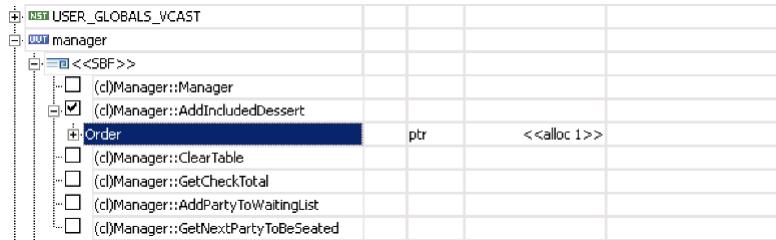
The parameter for **AddIncludedDessert** is displayed. It is a pointer to the array **Order**. We need to specify a value for **Dessert** for this stubbed subprogram to return to the test harness.

- To allocate one element of the array, enter 1 in the Input Values column for the pointer to **Order**.



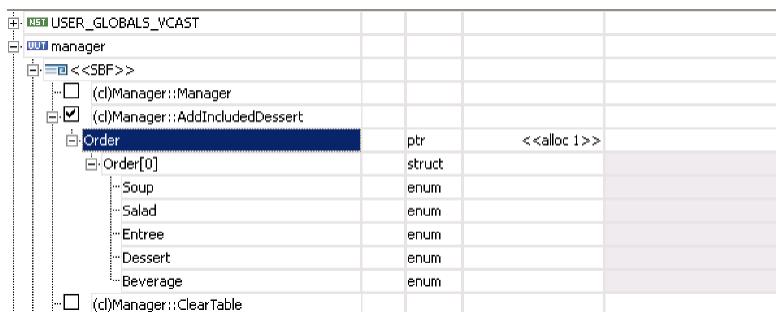
- Press Enter.

The text in the cell changes to “<<alloc 1>>” and a structure Order[0] is displayed.

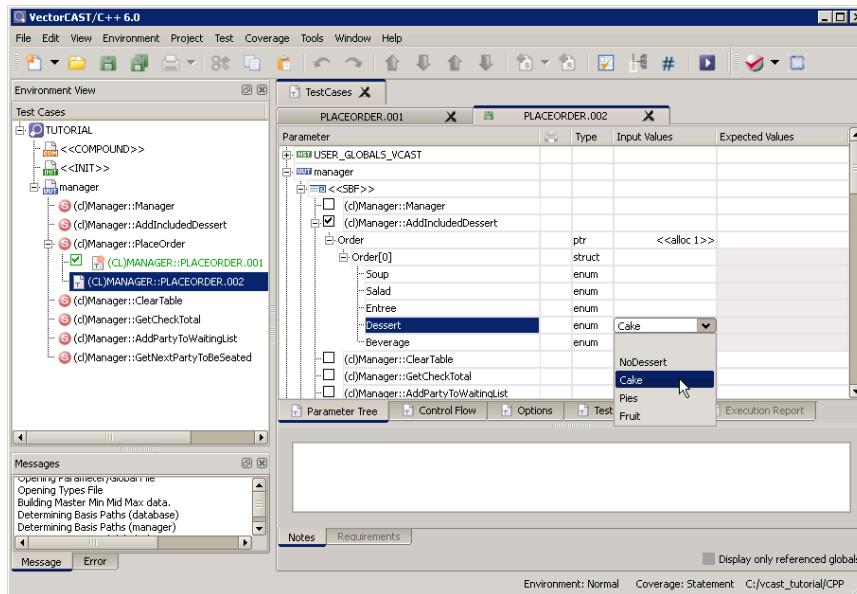


6. Click preceding Order[0].

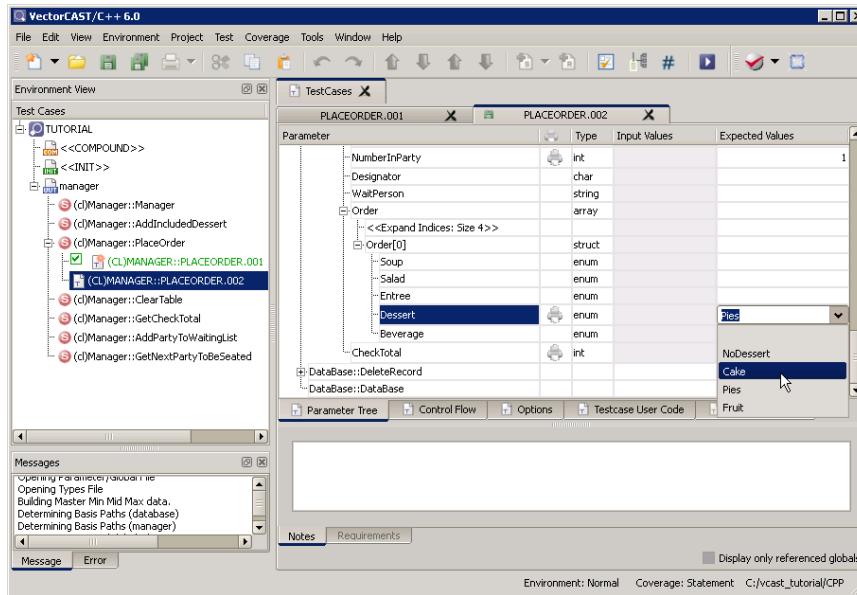
The enumerated elements of the structure are displayed.



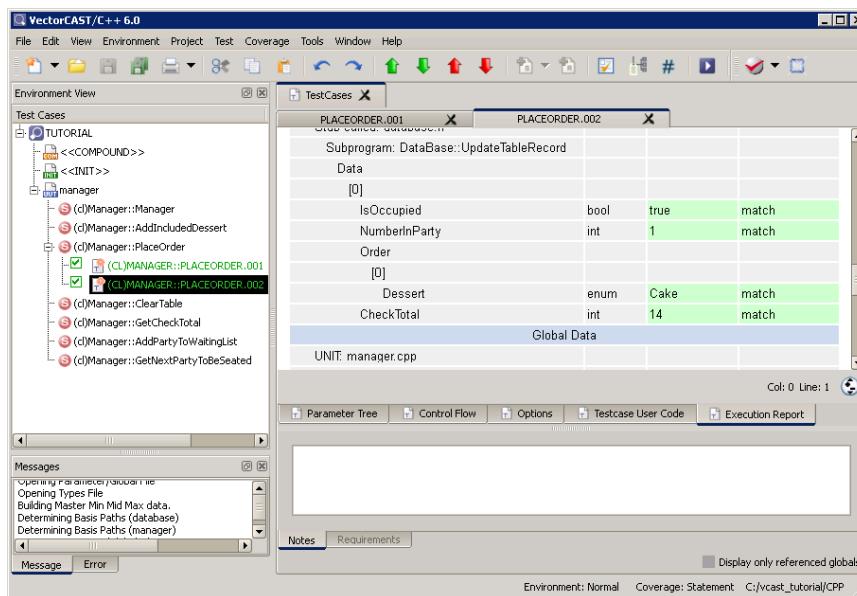
7. Set the Input Value for Dessert to **Cake**. When the test executes, Cake will be returned from **AddIncludedDessert**, instead of the non-stubbed return value, Pie.



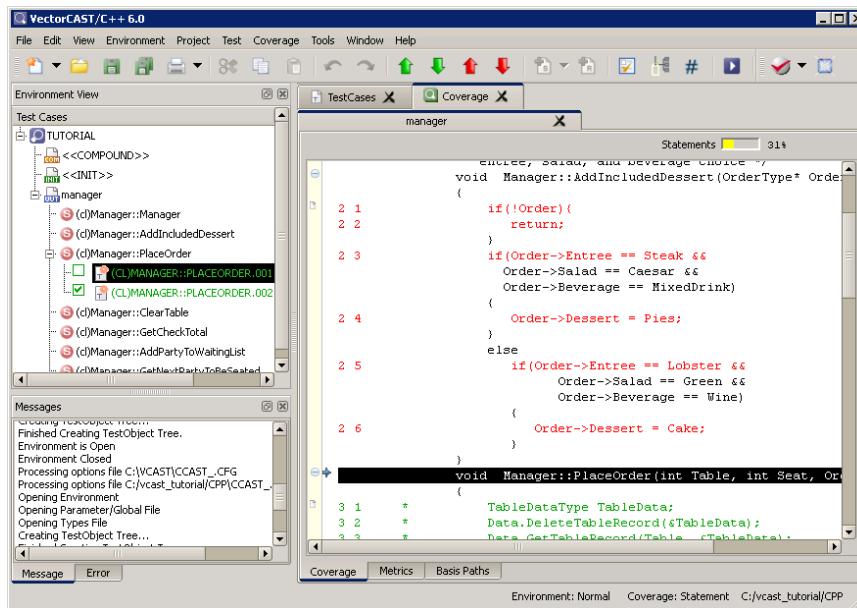
8. Scroll to the end of the Parameter Tree and locate **UpdateTableRecord**, which is part of Stubbed Subprograms.
9. Change the Expected Value of the Dessert from **PIE** to **CAKE**.



10. To save the test case, click the **Save** button on the Toolbar .
11. Execute the test case by clicking the **Execute** button .
12. The Execution Report tab comes to the top, displaying the results of the test execution. Scroll down to the end of the report to see that the expected value of Cake was matched.



13. To access the coverage achieved for this test case, click the green checkbox to the left of PLACEORDER.002.
An annotated version of the manager source code appears in the Coverage Viewer, showing the aggregate coverage achieved by both test cases.
14. Uncheck the green checkbox next to PLACEORDER.001.



In this test case, the subprogram `AddIncludedDessert` is not covered, as evidenced by statements 2.3 and 2.4 in red. The subprogram does not have coverage because it was stubbed during the execution of PLACEORDER.002.

15. In the Environment View, select the test case PLACEORDER.002, then choose **Test => Scripting => Export Script...**.
The Select Script Filename dialog appears.
16. Give the test script a name, such as `_CL_MANAGER__PLACEORDER.002.tst`, and click **Save**.
17. To open the test script, choose **File => Open**, set the File type to "Script Files (*.env *.tst)" and then select `_CL_MANAGER__PLACEORDER_002.tst`. Click **Open**.

The test script for _CL_MANAGER__PLACEORDER_002 looks like this:

```
-- VectorCAST 6.0 (01/05/12)
-- Test Case Script
--
-- Environment : TUTORIAL
-- Unit(s) Under Test: manager
--
-- Script Features
TEST.SCRIPT_FEATURE:C_DIRECT_ARRAY_INDEXING
TEST.SCRIPT_FEATURE:CPP_CLASS_OBJECT_REVISION
TEST.SCRIPT_FEATURE:MULTIPLE_UUT_SUPPORT
TEST.SCRIPT_FEATURE:STANDARD_SPACING_R2
TEST.SCRIPT_FEATURE:OVERLOADED_CONST_SUPPORT
TEST.SCRIPT_FEATURE:UNDERSCORE_NULLPTR
--
-- Unit: manager
-- Subprogram: (cl)Manager::PlaceOrder
-- Test Case: (CL)MANAGER::PLACEORDER.002
TEST.UNIT:manager
TEST.SUBPROGRAM:(cl)Manager::PlaceOrder
TEST.NEW
TEST.NAME:(CL)MANAGER::PLACEORDER.002
TEST.STUB:manager.(cl)Manager::AddIncludedDessert
TEST.VALUE:manager.<<GLOBAL>>. (cl).Manager.Manager.<<constructor>>.Manager()
().<<call>>:0
TEST.VALUE:manager.(cl)Manager::AddIncludedDessert.Order[0].Dessert:Cake
TEST.VALUE:manager.(cl)Manager::PlaceOrder.Table:2
TEST.VALUE:manager.(cl)Manager::PlaceOrder.Seat:0
TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Soup:Onion
TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Salad:Caesar
TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Entree:Steak
TEST.VALUE:manager.(cl)Manager::PlaceOrder.Order.Beverage:MixedDrink
TEST.VALUE:uut_prototype_stubs.DataBase::GetTableRecord.Data[0].NumberInParty:0
TEST.VALUE:uut_prototype_stubs.DataBase::GetTableRecord.Data[0].CheckTotal:0
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data
[0].IsOccupied:true
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data
[0].NumberInParty:1
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data[0].Order
[0].Dessert:Cake
TEST.EXPECTED:uut_prototype_stubs.DataBase::UpdateTableRecord.Data
[0].CheckTotal:12..16
TEST.END
```

The line beginning with TEST.STUB indicates that **AddIncludedDessert** should be stubbed during this test's execution. The TEST.VALUE line in bold specifies the Input Value that the stubbed subprogram returns to the test harness during this test's execution. This is possible only because manager is a Stubbed-by-Function (SBF) type of UUT, as specified during environment build.

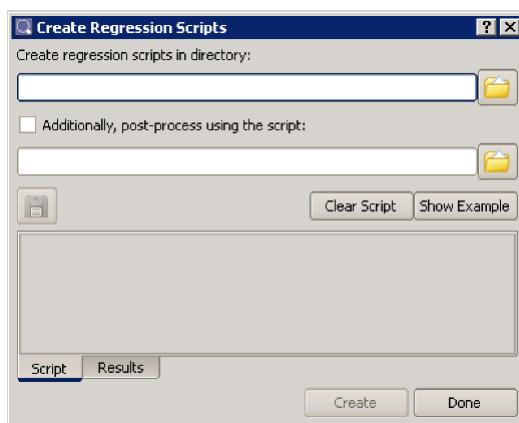
Creating Regression Scripts

Regression Scripts are the files needed to recreate the test environment at a later time. For a typical unit test environment, only 3 files are needed:

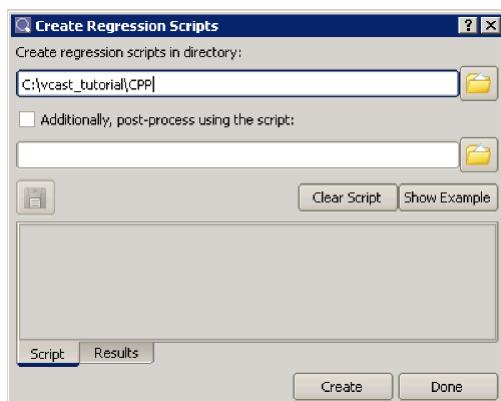
- > A shell script (.bat file on Windows, .sh or .csh on UNIX) sets the options, builds the environment, imports and executes the test cases, and generates reports
- > A test script (.tst) defines test cases to be imported by the shell script.
- > The environment script (.env) defines the settings for the environment, such as the UUT(s), which UUTs are Stub-by-Function, what is the method of stubbing, and where are the search directories.

1. Select **Environment => Create Regression Scripts....**

The Create Regression Scripts dialog appears.

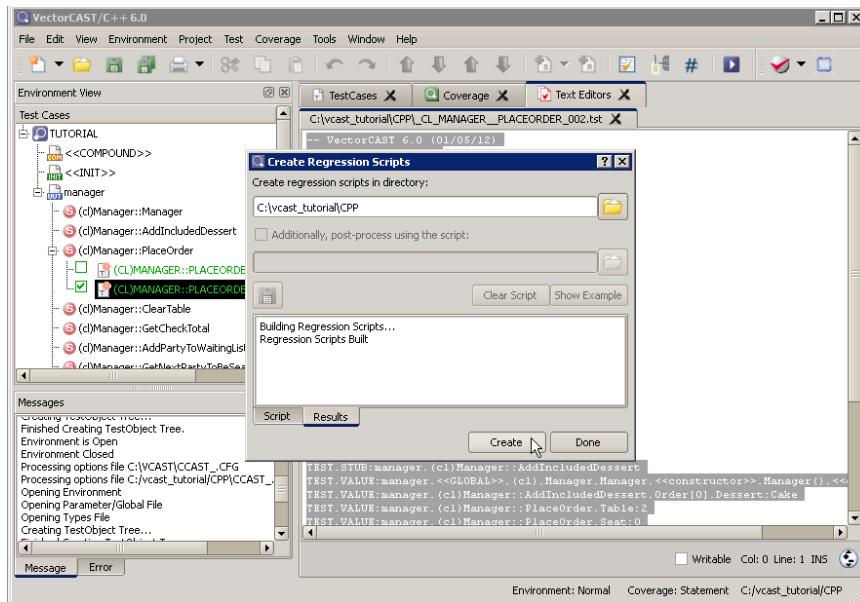


2. In the first edit box, click the **Browse** button and navigate to any directory you choose. For the purposes of this tutorial, we will specify the working directory as the destination for the regression scripts.



3. Click **Create**.

VectorCAST creates the three regression scripts in the location you specified.



- Click **Done** to close the dialog.

Now that you have the regression scripts, you can delete this environment and easily rebuild it at a later time simply by executing the shell script.

Tutorial Summary

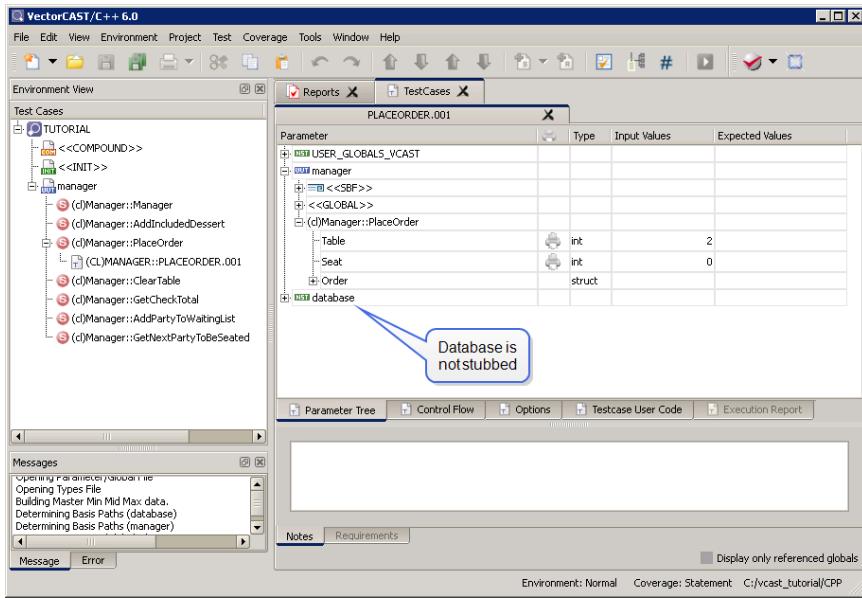
You have now completed the Basic Tutorial. You:

- > Built a test environment named **TUTORIAL** for testing a UUT named **manager**
- > Defined and executed a test case named **PLACEORDER.001**
- > Viewed source coverage information
- > Viewed the Test Case Management Report and the Aggregate Coverage Report for your test environment
- > Duplicated a test case
- > Stubbed a function in the UUT during test execution
- > Created regression scripts for this environment, which include the shell script, test script, and environment script

Troubleshooting

Problem: Dependents are not stubbed

If you do not see **GetTableRecord** (only **<<GLOBAL>>** and **TableData**, as shown below), then you did not stub database. As a result, the test harness is using the actual unit (database), which means you cannot specify any input or expected values for subprogram **GetTableRecord**.



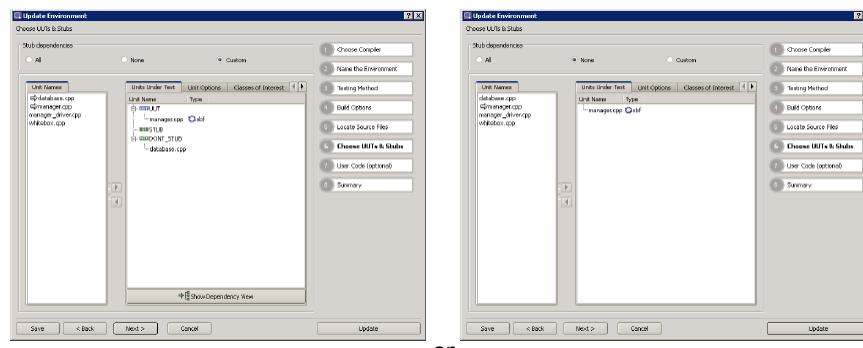
To solve this problem, you can update the environment to change database from not stubbed to stub:

1. Select **Environment => Update Environment**.

A dialog appears that looks similar to the Create New Environment dialog.

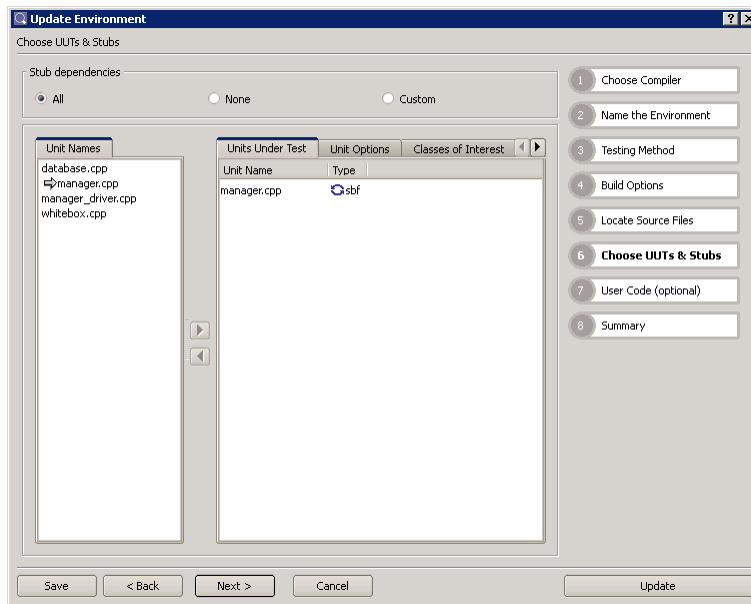
2. Click on **Choose UUTs & Stubs**.

Depending on whether you chose “Not Stubbed” when cycling through the types of stubbing or you selected the stub “None” radio button along the top, the Wizard looks like the following:



or

3. Click the radio button **All** under **Stub dependencies** to change **database** to be stubbed.



(Alternatively, you could have clicked **Custom**, then the toggle button  and then cycled the unit **database** until its type was **stub (by prototype)**).

Unit Name	Type
manager.cpp	sbf
database.cpp	stub (by prototype)

4. Click **Update**.

The environment rebuilds; **database** is now stubbed.

Problem: Input values entered instead of expected values

If you enter expected values into Input Test Data fields, you will see the test-case listing as shown below. Note that there is no information under **Expected Test Data**.

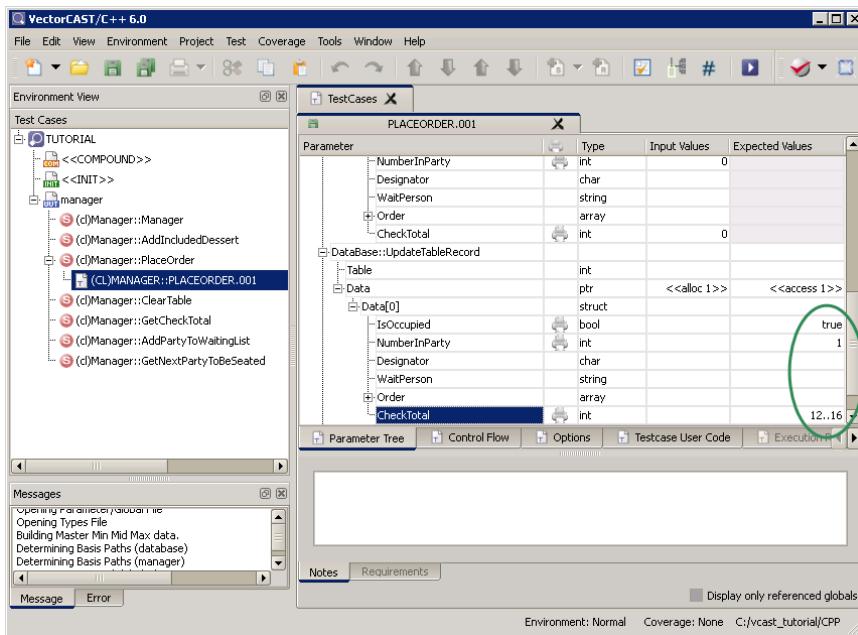
Input Test Data	
UUT:	manager.cpp
Subprogram:	Manager::PlaceOrder
Table	2
Seat	0
Order	
Soup	Onion
Salad	Caesar

Entree	Steak
Dessert	
Beverage	MixedDrink
Stubbed Subprograms:	
Unit: database.h	
Subprogram: DataBase::GetTableRecord	
Data	
[0]	
NumberInParty	0
CheckTotal	0
Subprogram:	
DataBase::UpdateTableRecord	
Data	
[0]	
IsOccupied	true
NumberInParty	1
CheckTotal	VARY FROM: 12.0000 To:16.0000 BY:1
Expected Test Data	

To solve this problem:

1. Select **File => Close** to close the test-case data report.
2. Double-click on (CL)MANAGER::PLACEORDER.001.
3. Delete the values **true**, **1**, and **12..16** from the Input Values column and enter them into the

Expected Values column (to the right of the Input column).



Multiple UUT Tutorial

This tutorial demonstrates how to use VectorCAST's multiple UUT feature to conduct integration testing.

When testing more than one UUT in an environment, it is typical that the units interact. VectorCAST's multiple UUT (multi UUT) feature enables you to do this.

In VectorCAST, there are two types of test cases: simple test cases and compound test cases. In the Basic Tutorial, you created and used a simple test case, which corresponded to a single invocation of a UUT. A compound test case is a collection of simple test cases that invokes a UUT in different contexts. The *data is persistent* across these invocations. Data persistence is very important when doing integration testing.

This tutorial will take you through the steps of building a compound test case to test the interaction between the unit **manager** and the unit **database**. The unit **manager** is a simple database in which actual data is stored in data objects defined within the unit. You are guided through the steps of building a compound test case to write data into this database. You then use the same compound test case to retrieve data from the database to verify it.

It is recommended that you review the relevant source listings in Appendix A, "Tutorial Source Code Listings" on page 311 before proceeding.

What You Will Accomplish

In this tutorial you will:

- > Build a multi UUT environment using the environment script from the Basic Tutorial as a starting point

- > Build a class constructor
- > Add a test case to unit **database**
- > Use the simple test cases to create a compound case
- > Verify the data flow between UUTs **manager** and **database**
- > Set processing iterations across ranges of input data
- > Build and execute a compound test case

Building a Multi UUT Environment

In this section, you will modify for integration testing the environment you built in the Basic Tutorial.

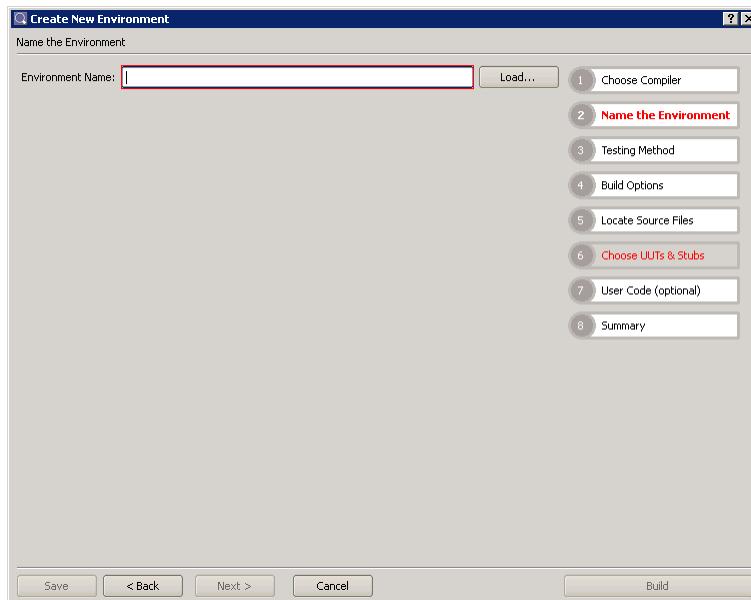
1. Start VectorCAST if it is not running.
If necessary, refer to "Starting VectorCAST" on page 8.

Loading an Environment Script

You will now change `database` from being a stubbed dependent in your test environment to being to a UUT:

1. Select **Environment => New => C/C++ Environment**.

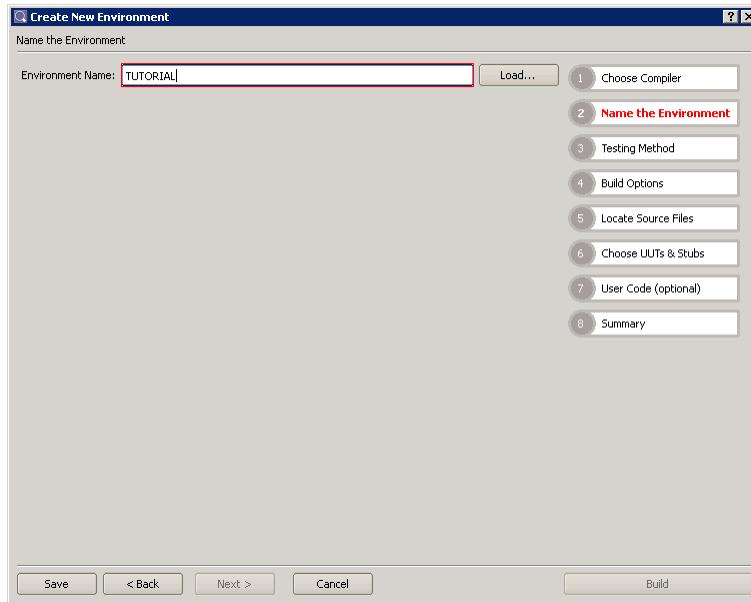
The Create New Environment wizard appears:



The “Choose Compiler” page is skipped because it has already been specified.

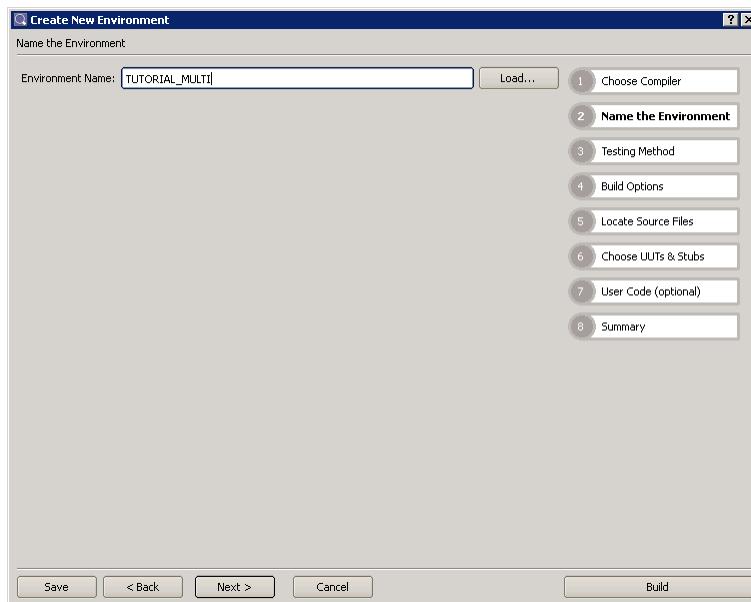
2. Click the **Load** button.

In the “Choose File” dialog, choose the file **TUTORIAL.env** and click **Open**.
The Wizard fills in with the settings from the environment script, **TUTORIAL.env**.

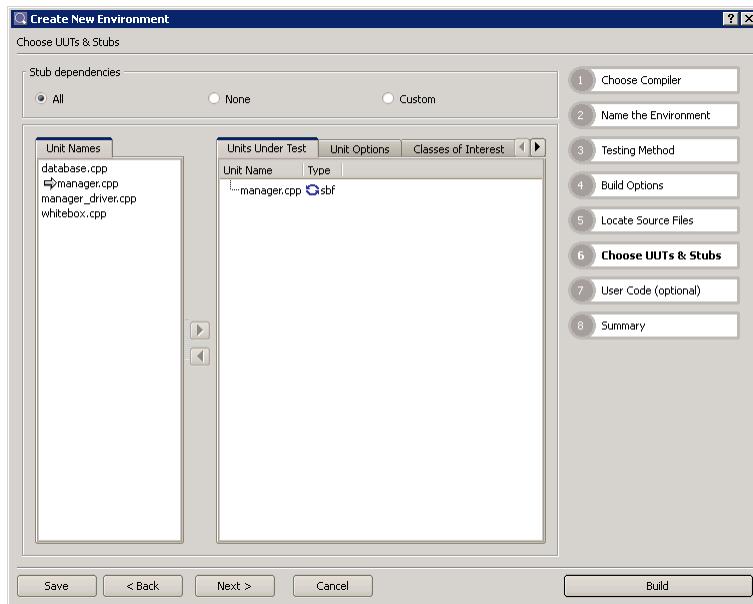


The name is outlined in red to indicate that it is not unique; there is already an environment of that name.

3. Change the name of this environment to “**TUTORIAL_MULTI**.”



4. Click **Choose UUTs & Stubs** (step 6).

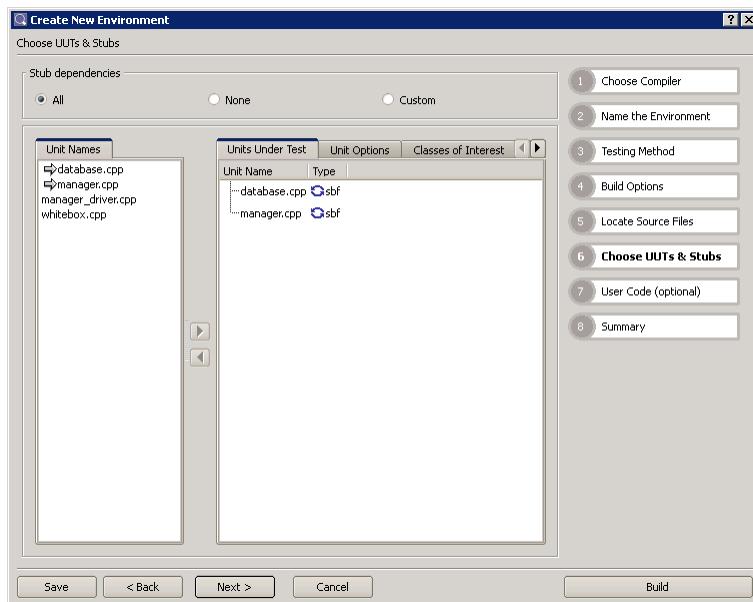


5. In the **Unit Names** pane, double-click **database.cpp**.



Note: Double-clicking a name in the Unit Names pane is a shortcut for selecting the name and then clicking the right-arrow (►).

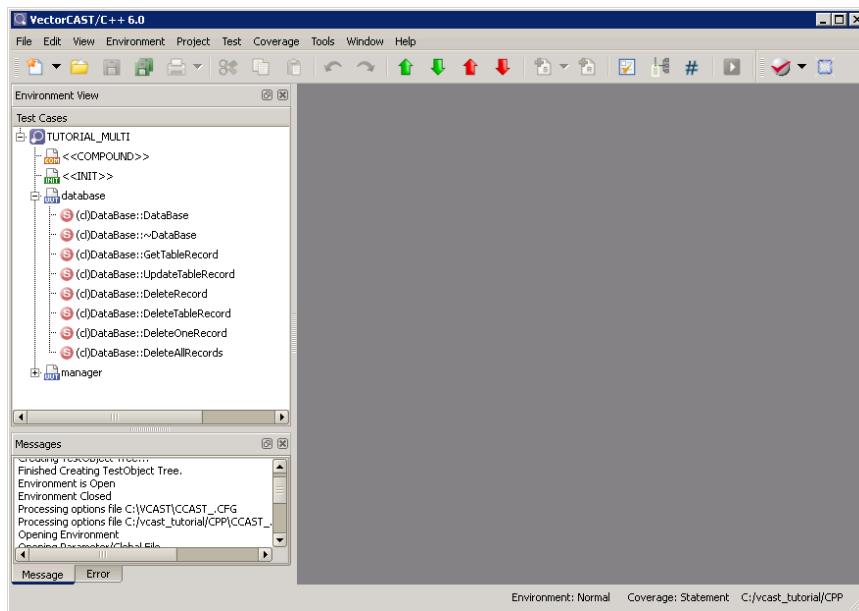
Both **database** and **manager** are now listed as Units Under Test with Stub-by-Function (SBF) enabled:



6. Click **Build**.

7. As the environment builds to include two UUTs, a series of messages is displayed in the Message window.

The environment is opened:



Note in the Environment View that database is now listed as a UUT, and also that it has several subprograms, including GetTableRecord and UpdateTableRecord.

In the next section, you will create a simple database test case and a simple manager test case and then combine these two simple cases to create a compound test case.

Building a Compound Test Case

You now have two UUTs to use toward creating a compound test case.

You create a compound test case by creating a separate test case for each subprogram to be included in the test and then combining these test cases into a compound test case.

To create the simple test cases, you use the same procedure you used in the Basic Tutorial to create the test case (PLACEORDER.001) that you deleted for the purposes of this tutorial.

Build a Test Case for the Constructor

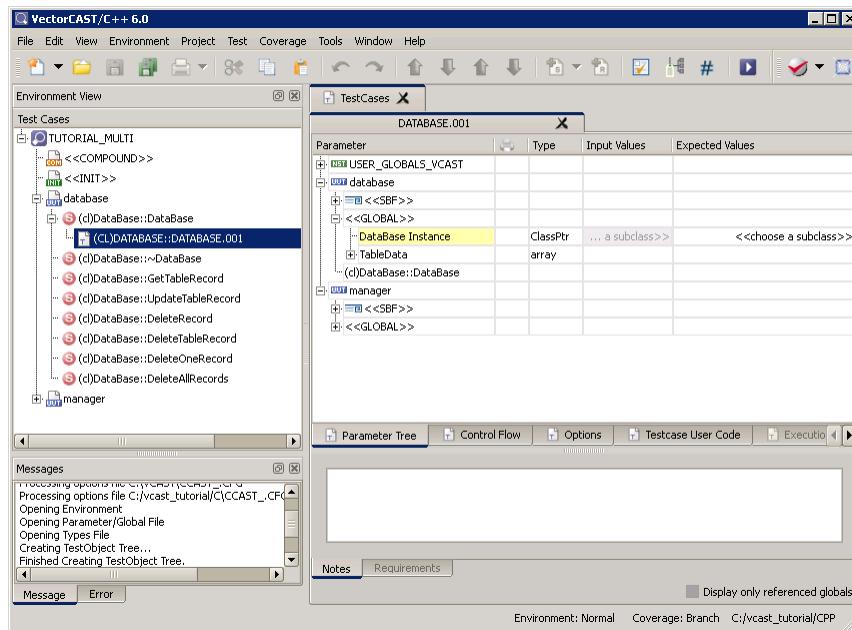
When building a compound test case for a C++ class, you should construct an instance of the class as the first slot in the compound test. Doing this enables all slots to share the same class object, and thus member variables are shared between test cases.

In this section, you will:

- > Create a test case to create an instance of the database class which will be shared; a new PLACEORDER.001 test case will create the instance of the manager class.
- > Create and add input values to the test case PLACEORDER.001, just as you did in the Basic Tutorial. This test case will place an order at a table in the restaurant.

- > Create a simple test case for subprogram GetTableRecord, which reads data from the database. This test case will be used to verify that PlaceOrder is updating the database properly.
- > Combine these test cases into a compound test case

1. Right-click on **(cl)DataBase::DataBase** and insert a new test case. The name defaults to **(CL)DATABASE::DATABASE.001**.
2. Under the UUT **database**, expand **<<GLOBAL>>**.



The yellow highlighting indicates the class that is associated with the subprogram under test. There may be many classes defined in a unit, and the highlighting indicates which one you would want to choose a subclass and constructor for.

A special case is the test case for a constructor itself. You don't need to pick a subclass and constructor, so it is dimmed. When VectorCAST executes the test case, the constructor is automatically called, and the class is therefore automatically instantiated.

At this point you have constructed a class object for the DataBase class. When you build the compound test case, this test case should be placed first, to "initialize" a class object so that all of the methods of the class can be called within the context of the same class object.

Insert PLACEORDER.001

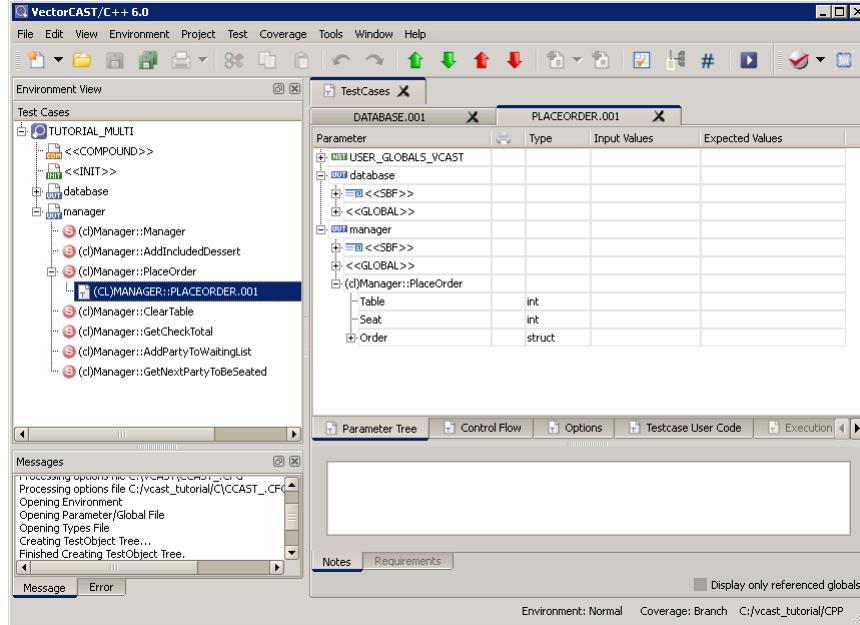
In this section, you will:

- > Create and add Input Values to the test case PLACEORDER.001. This test case will place an order at a table in the restaurant.
- > Create a simple test case for the subprogram GetTableRecord, which retrieves data from the database. This test case will be used to verify that PlaceOrder is updating the database properly.
- > Place these test cases into a compound test case

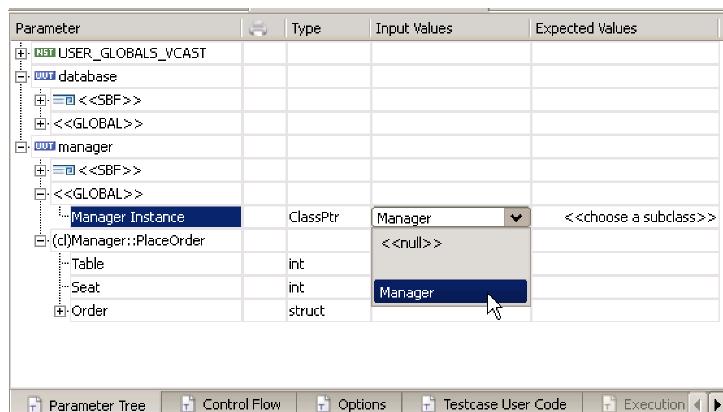
To create and add order data to PLACEORDER.001:

1. In the Environment View, expand **manager** into its subprograms, and then expand **(cl) Manager::PlaceOrder** into its subprograms.
2. Right-click **PlaceOrder** and choose **Insert Test Case**.

The test case PLACEORDER.001 displays in the Test Case Editor.



3. Under **manager**, expand **<<GLOBAL>>**.
4. Next to the yellow-shaded class instance, Manager Instance, click the Input Values column cell containing **<<choose a subclass>>**, and select **Manager** from the drop-down list.
5. In the Input Values column for the subclass Manager, click the Input Values cell containing **<<choose a constructor>>**, and select **Manager::Manager()** from the drop-down list. Because the example source code has only one constructor defined, it is automatically selected.



6. Scroll down and locate the subprogram **(cl)Manager::PlaceOrder**. Assign Input Values for Table and Seat as you did in the Basic Tutorial:

Table **2**

Seat **0**

7. Expand **Order**.

8. Assign values to the Order parameters as indicated below.

The idea is to populate an Order record with a set of Input Values you will also use as expected values for a retrieval operation involving GetTableRecord:

Soup **ONION**

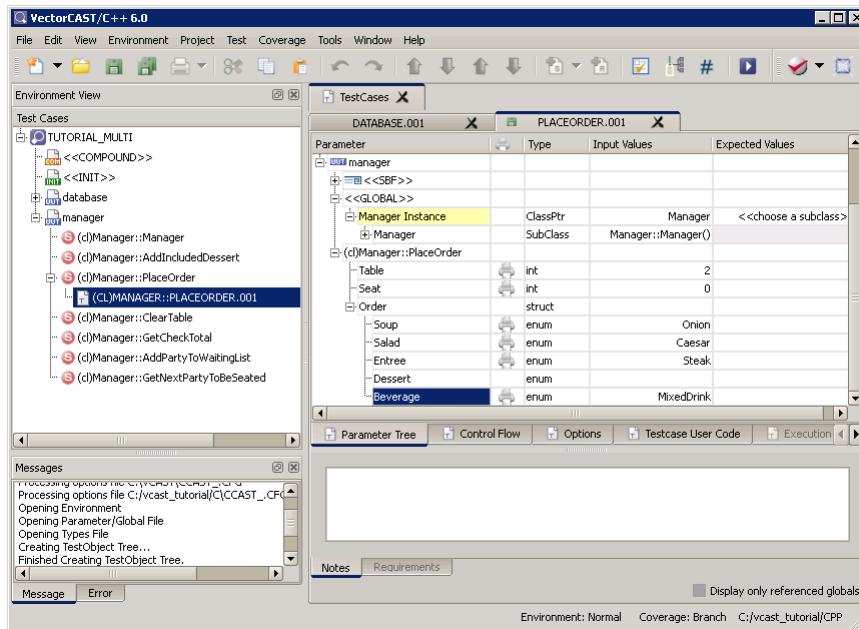
Salad **CAESAR**

Entree **STEAK**

Dessert **(leave unspecified)**

Beverage **MIXED_DRINK**

These values translate into the following situation: Someone is occupying seat 0 at table 2, and has ordered onion soup, a Caesar salad, a steak entrée, a mixed drink. This combination yields an included dessert of Pie, so we will expect PIE when later verifying the Order.



9. Save the test case by clicking the **Save** button on the Toolbar .

Because you will be making your test case part of a compound test case, and you only want to use this test within a compound case, you need to designate it as Compound Only. Compound Only test cases can only be executed from within a compound test. They cannot be executed individually or as part of a batch.

In some cases, you will want to use simple test cases both individually and as part of a compound

case. In these instances, it is not necessary to apply the Compound Only attribute to the simple cases.

10. Right-click **(CL)MANAGER::PLACEORDER.001** and select **Compound Only**.

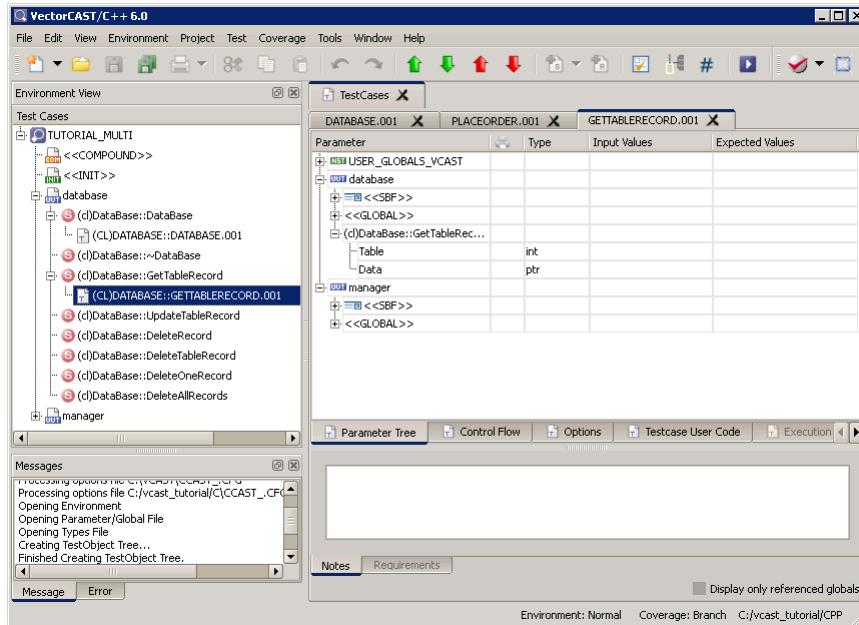
The test case typeface changes to *italic*, signifying that this test case has been designated Compound Only.

Create a Test Case for the GetTableRecord Function

In this section, you will create a test case for the function GetTableRecord. This function retrieves data from the database and verifies it against a set of expected values.

1. In the Environment View, expand **database** into its subprograms (if not already expanded).
2. Right-click **(cl)DataBase::GetTableRecord** and select **Insert Test Case**.

The test case **GETTABLERECORD.001** appears in the Test Case Editor:



We do not want to set the class instance in this test case, because doing so would overwrite the class instantiated by the constructor test case.

3. Keeping with our example situation, enter **2** as the input value for **Table**.



Note: Make sure you enter the value into the Input Values column, and not into the Expected Values column. You will enter the expected values for this case in the next step.

4. To allocate one element of the array Data, enter **1** in the Input Values column for **Data**, or double-click **Data**.

UUT database				
+ <<SBF>>				
+ <<GLOBAL>>				
- (c) DataBase::GetTableRec...				
Table	int		2	
Data	ptr	[1]		
+ UUT manager				

One element named Data[0] appears, and the "1" is replaced with "<<alloc 1>>".

UUT database				
+ <<SBF>>				
+ <<GLOBAL>>				
- (c) DataBase::GetTableRec...				
Table	int		2	
Data	ptr		<<alloc 1>>	
Data[0]	struct			
+ UUT manager				

5. Click the Expected Values column for **Data** and enter **1**, to gain access to the structure.
The cell displays "<<access 1>>".
6. Click the preceding **Data[0]**.
Data[0] expands into its components.

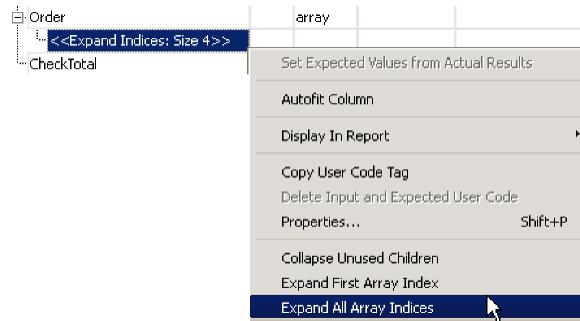
UUT database				
+ <<SBF>>				
+ <<GLOBAL>>				
- (c) DataBase::GetTableRec...				
Table	int		2	
Data	ptr		<<alloc 1>>	<<access 1>>
Data[0]	struct			
IsOccupied	bool			
NumberInParty	int			
Designator	char			
WaitPerson	string			
+ Order	array			
CheckTotal	int			
+ UUT manager				

7. In the **Expected Values** column, enter expected results as follows:
IsOccupied **true**
NumberInParty **1**

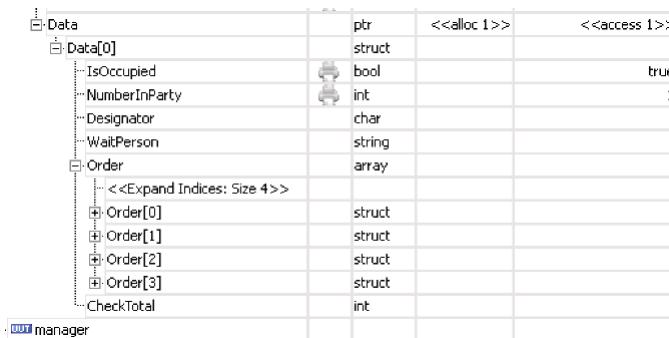


Note: Make sure you enter these values in the Expected Values column.

8. Expand **Order**.
Order is an array of structures of size 4.
9. To view the array-indices of Order, right-click on **<<Expand Indices: Size 4>>** and select **Expand All Array Indices**.



The array expands, displaying the four array elements:



In this example, you want to verify input values written to the application database. The data consists of an order taken for seat 0 at table 2.

10. Expand **Order[0]**.

Order[0] corresponds to seat 0. The expected values you specify for Order[0] must match the input values you previously specified in PLACEORDER.001.



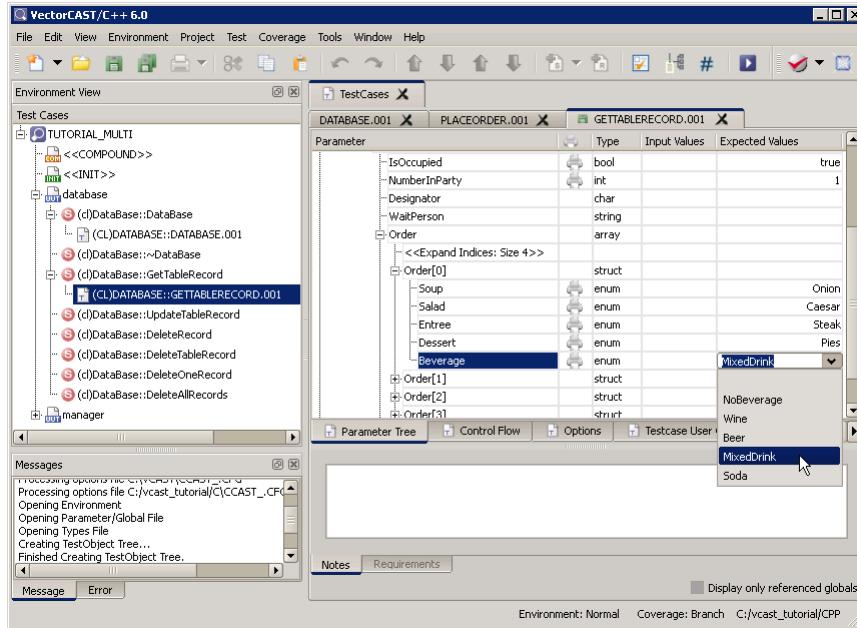
Important: If you use any other index, the expected results will not match.

11. Assign the following **Expected Values** to the appropriate fields of **Order[0]**:

Soup	Onion
Salad	Caesar
Entree	Steak
Dessert	Pies
Beverage	MixedDrink



Note: Make sure you enter these values into the **Expected Values** column. If the Expected Values column is dimmed, then you forgot to “Click the Expected Values column for **Data** and enter **1**, to gain access to the structure.”



12. Click **Save** button

Note that Order[1], Order[2], and Order[3] are not populated with data. Populating a single Order record is sufficient to verify the operation of the two subprograms under test.

13. Right-click **(CL)DATABASE::GETTABLERECORD.001** in the Environment View, and then select **Compound Only**.
14. While you're at it, right-click on **(CL)DATABASE::DATABASE.001** as well, and choose **Compound Only**.

You now have three test cases built: The first, DATABASE.001, is the constructor for the DataBase instance. We will use this test case in the Compound test case to create a common instance. The second case (PLACEORDER.001) places a dinner order from seat 0 at table 2. This order is stored in the database by way of `PlaceOrder` calling `UpdateTableRecord`. The third case (GETTABLERECORD.001) retrieves the order from the database for verification.

You will now use these test cases to create a compound test case.

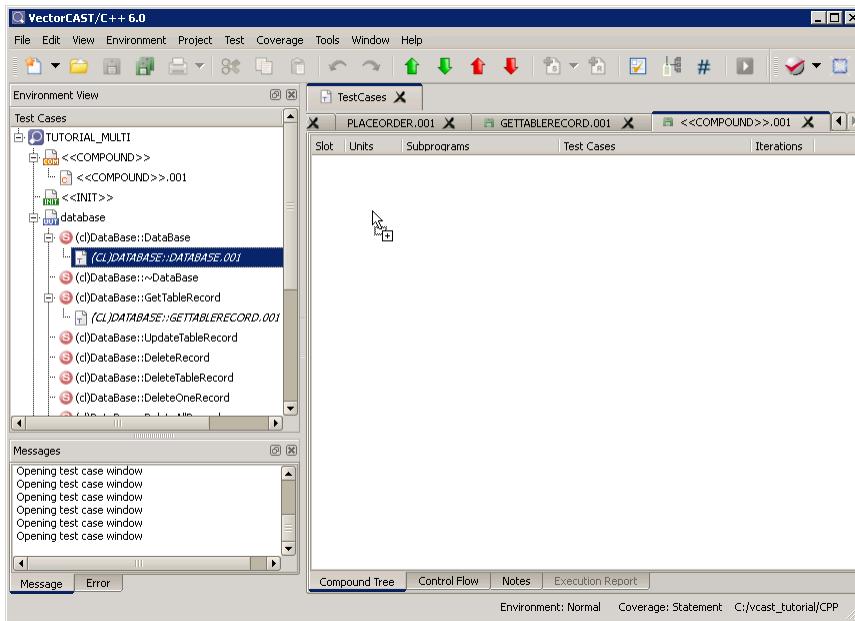
Creating a Compound Test Case

If you were to execute your test cases individually, they would fail, because the test harness would be called separately for each test case, and the data would not be retained in the data structure between the two calls. In order for your test cases to work together, they must be combined into a compound case.

1. In the Environment View, right-click **<<COMPOUND>>** and select **Insert Test Case**. A test case named **<<COMPOUND>>.001** appears under **<<COMPOUND>>**, and is selected (highlighted) by default.
2. Click and hold the cursor on **(CL)DATABASE::DATABASE.001** in the Environment View. Drag until the



symbol appears in the Test Case Editor; release.



An entry appears in the Compound Tree as slot 1.

3. Drag and drop **PLACEORDER.001** into the Compound too.
4. Do the same with test case **GETTABLERECORD.001**.

Your compound test case has three slots: DATABASE.001, PLACEORDER.001 and GETTABLERECORD.001:

Slot	Units	Subprograms	Test Cases	Iterations
1	database	(c)DataBase::DataB...	(CL)DATABASE::DATABASE.001	1
2	manager	(c)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.001	1
3	database	(c)DataBase::GetTableRecord	(CL)DATABASE::GETTABLERECORD.001	1

5. Save.

You are now ready to execute your compound test case.

6. In the Environment View, select **<<COMPOUND>>.001**, then click Execute Test  on the toolbar.

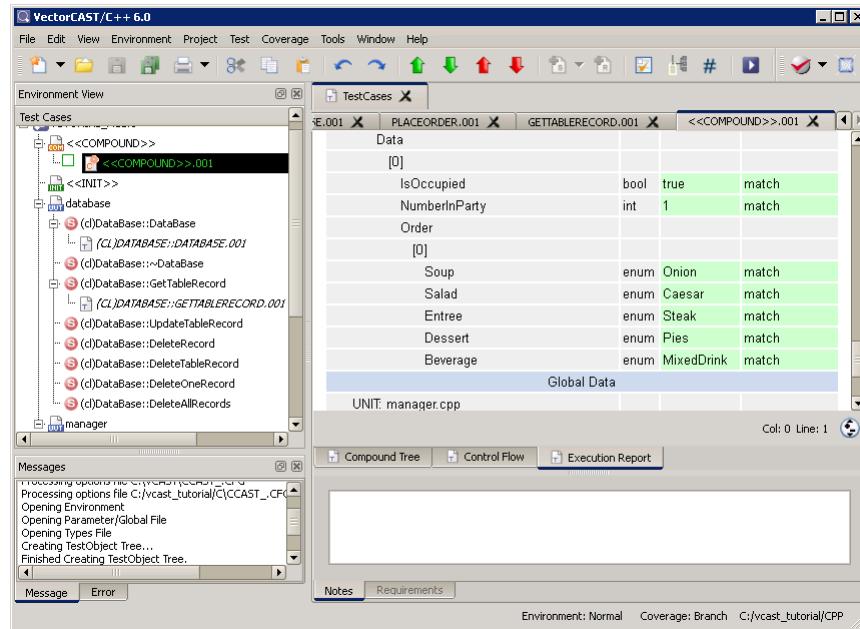
An execution report for the compound test case appears in the Report Viewer.



Note: If your compound test case fails, make sure PLACEORDER.001 uses parameter values table 2, seat 0, and that GETTABLERECORD.001 uses table 2 and the proper expected values in Order[0], and both <<alloc 1>> and <<access 1>>.

7. Scroll down the report.

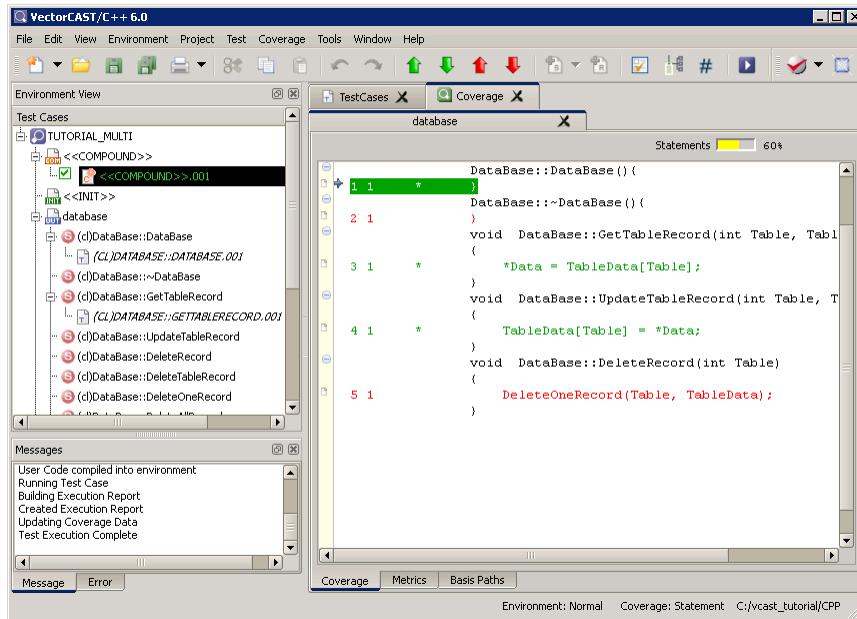
Note that the expected values you specified were matched against corresponding actual values, verifying the data flow from manager to database.



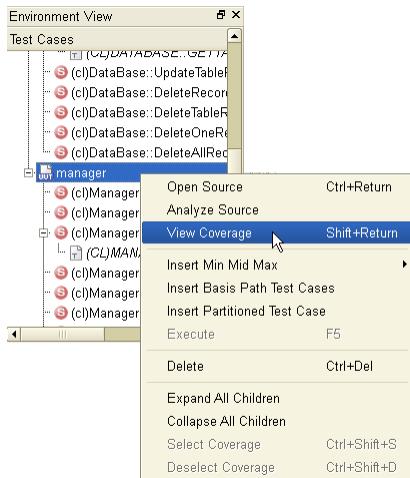
- Click the checkbox preceding <<COMPOUND>>.001 in the Environment View:



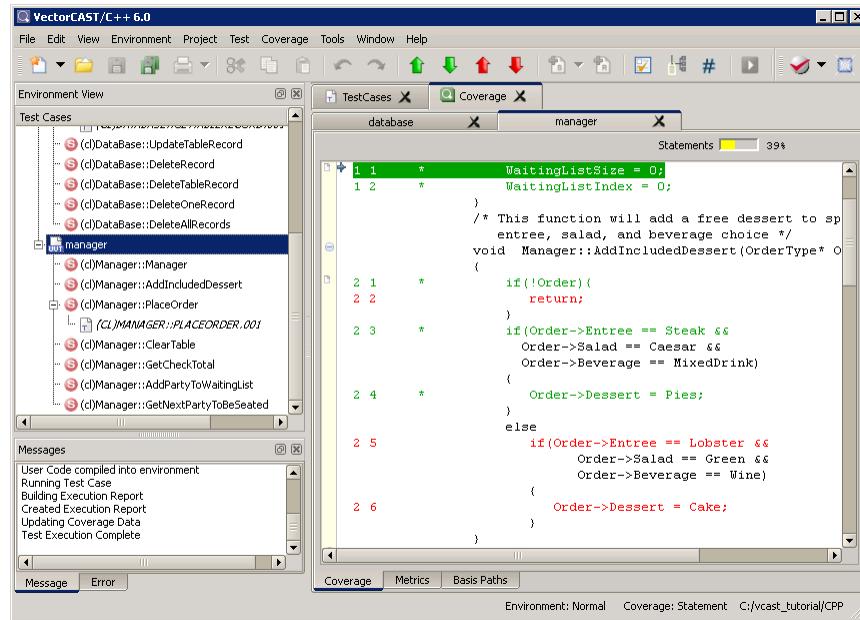
The tab for database appears in the Coverage Viewer because this is the unit called in the first slot.



- To view the coverage for the unit manager, right-click unit **manager** in the Environment View, and choose **View Coverage**.



A tab for manager opens in the Coverage Viewer, scrolled to the beginning of the subprogram PlaceOrder.



In the Coverage Viewer, note that the statement assigning the value Pie to Order-> Dessert is green, indicating that it was covered by test execution.

Initializing Branch Coverage

To find out which branches are covered by executing this compound test case:

1. Choose **Coverage => Initialize => Branch**.

The status bar shows “Coverage: Branch” and the checkbox next to <<COMPOUND>>.001 is removed, because the statement results were made invalid during coverage initialization. To see branch coverage achieved, you must execute the test case again.

Note that the *execution results* were not made invalid during coverage initialization.

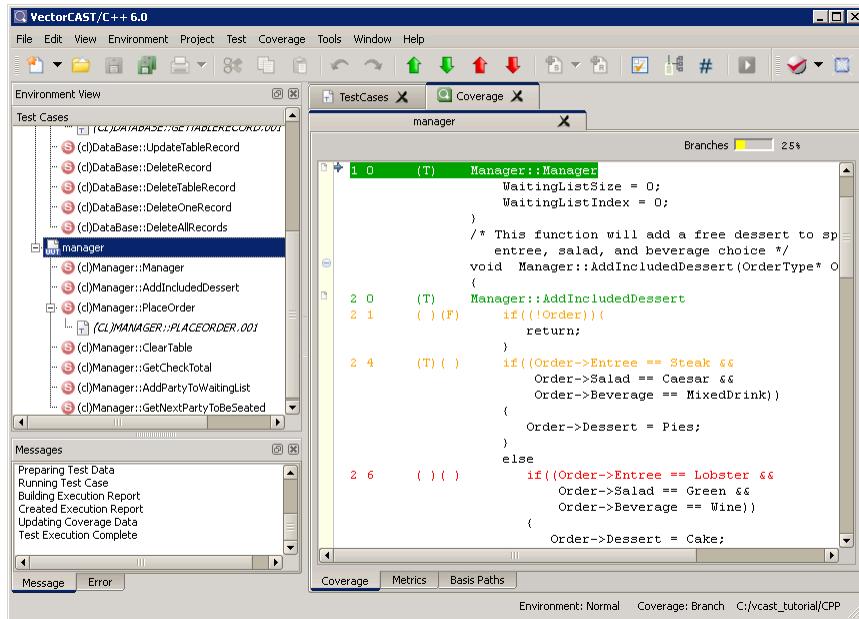
2. Select <<COMPOUND>>.001 in the Environment View and click the **Execute** button in the toolbar.
3. Click the checkbox preceding <<COMPOUND>>.001 in the Environment View:



4. Open the Coverage Viewer for the manager unit by right-clicking **manager** in the Environment View and choosing **View Coverage**.
5. Scroll to the first covered line of PlaceOrder.

In the Coverage Viewer, the instances of the '(T)' symbol indicate that two PlaceOrder branches were covered: the first at the subprogram entry point; the second at the Steak branch in

the `switch` statement.



VectorCAST allows you to build more complex tests by linking multiple simple test cases together. This feature allows you to test threads of execution while maintaining the data common to the various units and subprograms.

Adding Test Iterations for Fuller Coverage

You can reach fuller coverage of `manager` by having each seat at table 2 order a different entrée while keeping the other order items constant. Varying the Entree value ensures that each branch in the `switch` statement will be exercised by the test.

In this section, you will modify test case `PLACE_ORDER.001` to iterate over the 4 seats at table 2 while iterating over the four entrées. The net effect will be to assign a different entrée to each seat.

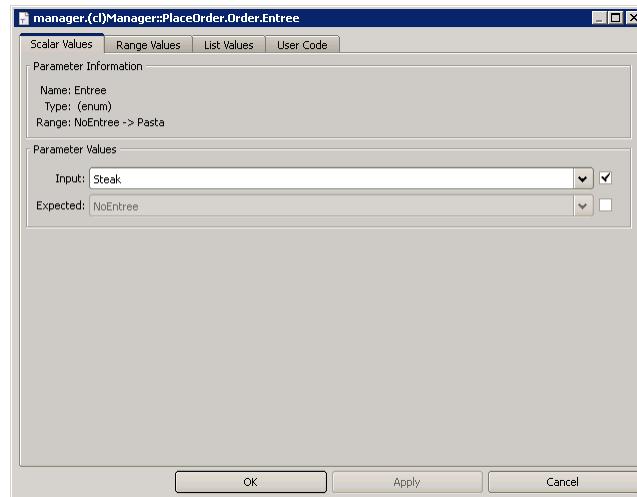
1. In the Environment View, double-click **PLACEORDER.001**.
The test case `PLACEORDER.001` opens in the Test Case Editor.
2. Enter **0..3** as the input range for **Seat**; press **Enter**.

Your entry changes to 0..3/1. The /1 indicates that the input value for Seat will iterate over its range with a delta of 1. (It is possible to specify a delta other than 1.)

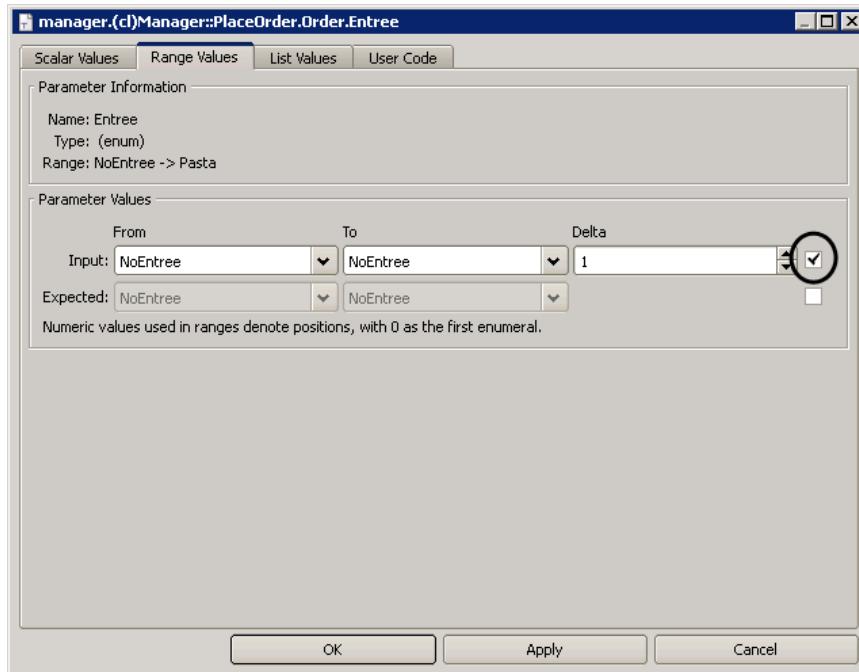
manager			
<<SBF>>			
<<GLOBAL>>			
Manager Instance	ClassPtr	Manager	<<choose a subclass>>
+ Manager	SubClass	Manager::Manager()	
(c)Manager::PlaceOrder			
Table	int	2	
Seat	int	0..3/1	
Order	struct		
Soup	enum	Onion	
Salad	enum	Caesar	
Entree	enum	Steak	
Dessert	enum		
Beverage	enum	MixedDrink	

3. Double-click **Entree**.

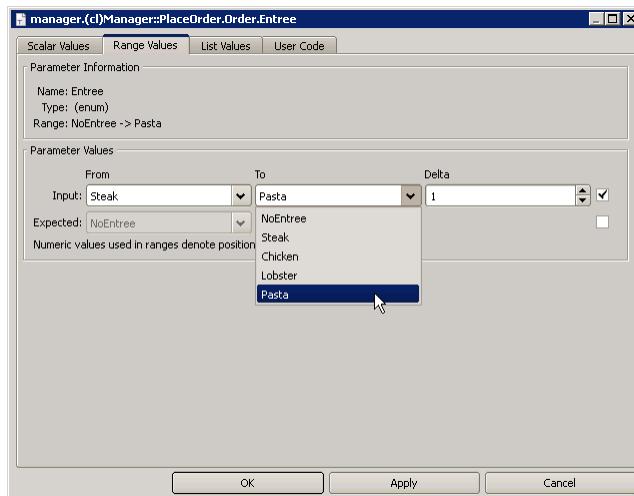
A dialog appears for setting parameter specifications:



4. Click the **Range Values** tab, and then enable the **Input** fields by clicking the checkbox at the far right:



5. Select **Steak** for **From**; **Pasta** for **To**; leave **Delta** at **1**:



6. Click **OK**.

UUT manager				
<<SBF>>				
<<GLOBAL>>				
Manager Instance	ClassPtr	Manager	<<choose a subclass>>	
Manager	SubClass		Manager::Manager()	
(d) Manager::PlaceOrder				
Table	int	2		
Seat	int	0..3/1		
Order	struct			
Soup	enum	Onion		
Salad	enum	Caesar		
Entree	enum	Steak..Pasta/1		
Dessert	enum			
Beverage	enum	MixedDrink		

7. Click **Save**.

You have now assigned input ranges to two parameters. By default, when two parameters are assigned input ranges, they will iterate in *parallel*, which is what you want to occur in this case. (There is a setting to make them iterate in combination.)

These two parameters will iterate with the following value pairs:

Seat = 0 Entree = Steak

Seat = 1 Entree = Chicken

Seat = 2 Entree = Lobster

Seat = 3 Entree = Pasta

Given this information, you can specify expected values in your compound test case for each seat.

8. Click the tab for test case **GETTABLERECORD.001** at the top of the Test Case Editor.

The parameter tree for test case GETTABLERECORD.001 comes to the top. You need to edit this tree to reflect the expected results for your compound test.

You now need to specify expected values for each of the other three seats at table 2.

9. You have already specified values for Order[0], so move on to Order[1].

10. Expand **Order[1]**. (If you have recently re-opened the TUTORIAL_MULTI environment, then the unused array elements have been collapsed. If Order[1] is not visible, simply right-click on **<<Expand Indices: Size 4>>** and choose **Expand All Array Indices**.)

11. Order[1] maps to seat 1. Specify the following *expected values* for seat 0:

Soup Onion

Salad Caesar

Entree Chicken

Dessert NoDessert (see Note)

Beverage MixedDrink



Note: For Order[1], Order[2], and Order[3], PlaceOrder calls AddIncludedDessert with a soup, salad, and entrée combination that does not qualify for Pie, so we expect NoDessert.

12. Expand **Order[2]**.

Enter the following expected values for seat 2:

Soup Onion

Salad	Caesar
Entree	Lobster
Dessert	NoDessert
Beverage	MixedDrink

13. Expand **Order[3]**.

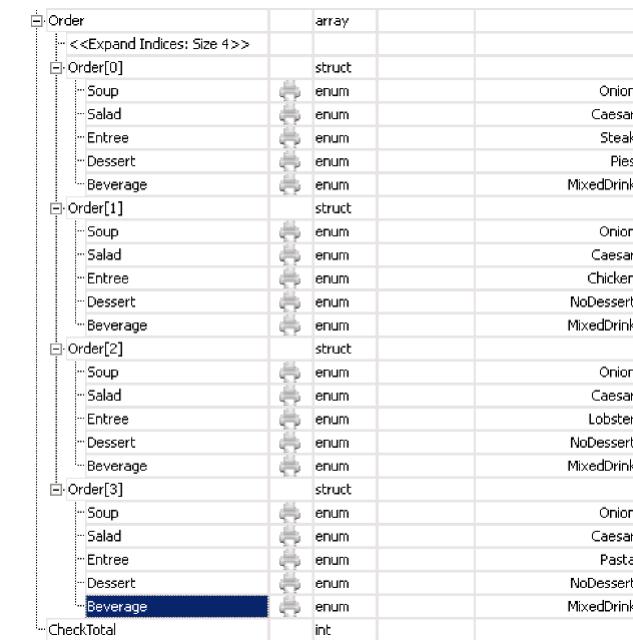
Enter the following expected values for seat 3:

Soup	Onion
Salad	Caesar
Entree	Pasta
Dessert	NoDessert
Beverage	MixedDrink



Note: Be sure you have populated the Expected Values column for Order, and not the Input Values column.

You now have Order data for each seat at table 2. Note that these orders are identical in each case except for the entrée and dessert. Only the Entree variable is needed to exercise each branch in the switch statement.

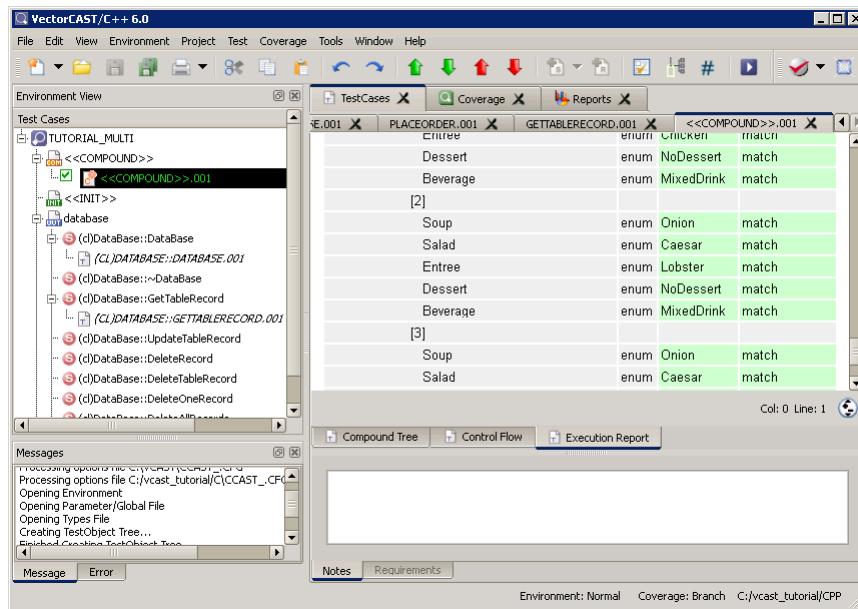


You must now change the number of occupied seats at table 2 to reflect the current test case.

14. Scroll up to **NumberInParty** and change the expected value to **4**.
15. Click Save.
16. Select **<<COMPOUND>>.001** in the Environment View, then click the **Execute** button . **<<COMPOUND>>.001** turns green; an execution report for your compound test appears in the

Test Case Editor. Your test has passed.

17. Scroll down the report to view the green-coded results for each element of Order:



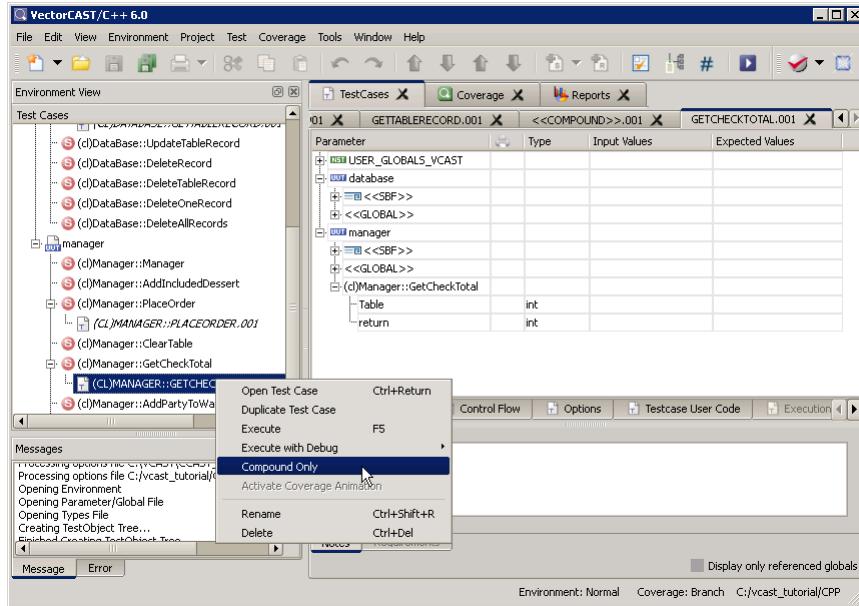
Adding Test Cases to a Compound Test

In this section, you will verify the check total for the four entrée orders at table 2 by adding a third test case to your compound case.



Note: The amount of instruction given in this section will be a little less than in the previous sections.

1. Under the manager unit, insert a test case for (cl)Manager::GetCheckTotal, and make it Compound Only.

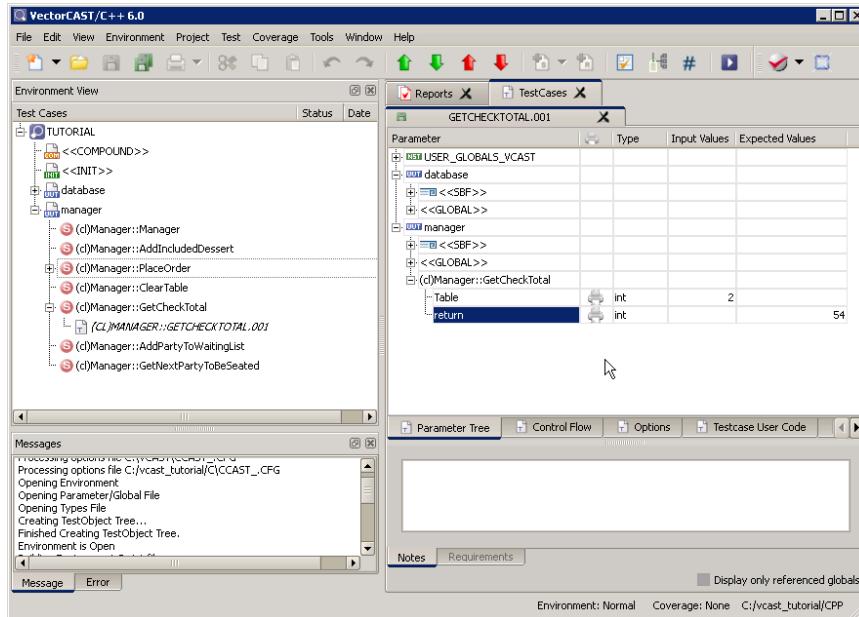


2. Enter **2** as the *input value* (*not expected value*) for **Table**.

On the basis of the entrée prices coded into the application, you expect the check to total \$54 (for one of each entrée).

3. Enter **54** as the *expected value* for **return**.

4. Save.



You can now add your new test case to the compound case.

5. Double-click **<>COMPOUND>>.001** in the Environment View.

- Click the **Compound Tree** tab at the bottom of the Test Case Editor.

The Test Case Editor displays the slots in test case <<COMPOUND>>.001.

- Add test case **GETCHECKTOTAL.001** to the compound by dragging and dropping it as the last slot.

GETCHECKTOTAL.001 displays as slot 4 in your compound test case:

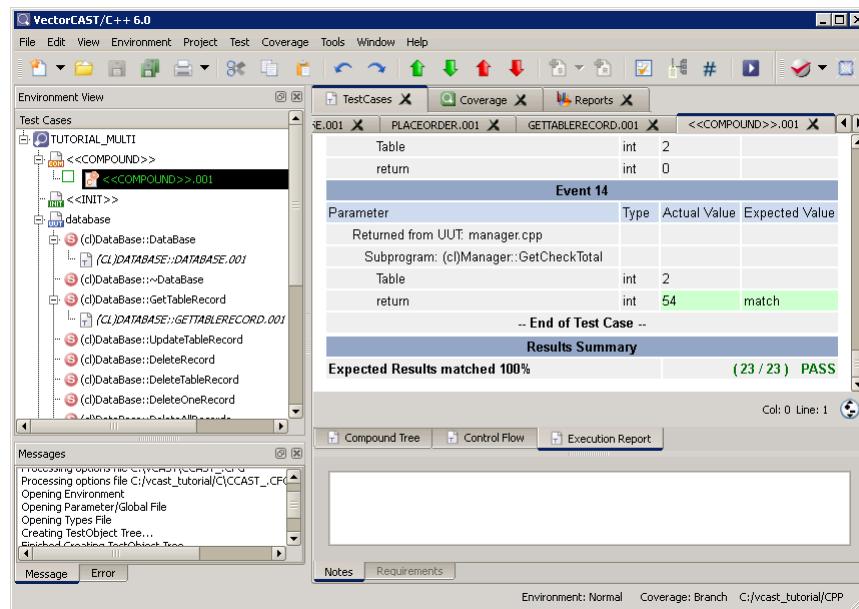
Slot	Units	Subprograms	Test Cases	Iterations
1	database	(d)DataBase:: DataBase	(CL)DATABASE::DATABASE.001	1
2	manager	(d)Manager::PlaceOrder	(CL)MANAGER::PLACEORDER.001	1
3	database	(d)DataBase::GetTableRecord	(CL)DATABASE::GETTABLERECORD.001	1
4	manager	(d)Manager::GetCheckTotal	(CL)MANAGER::GETCHECKTOTAL.001	1

- Save.

- Select <<COMPOUND>>.001 in the Environment View, then click to execute it.

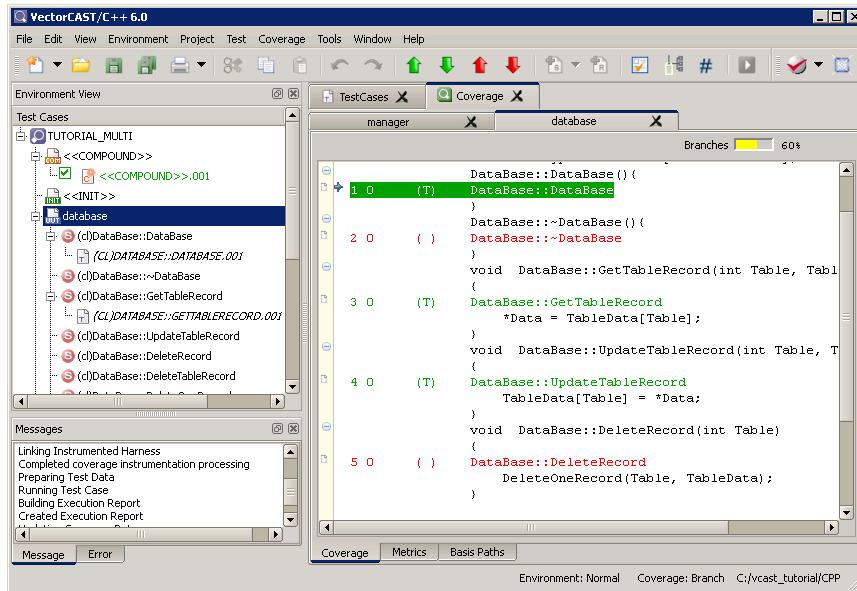
<<COMPOUND>>.001 turns green in the Environment View; and an execution report displays in the Test Case Editor.

- Click the green down-arrow on the toolbar, or scroll down the report until you come to the green-shaded results section. You can see that the check total for table 2 was \$54.

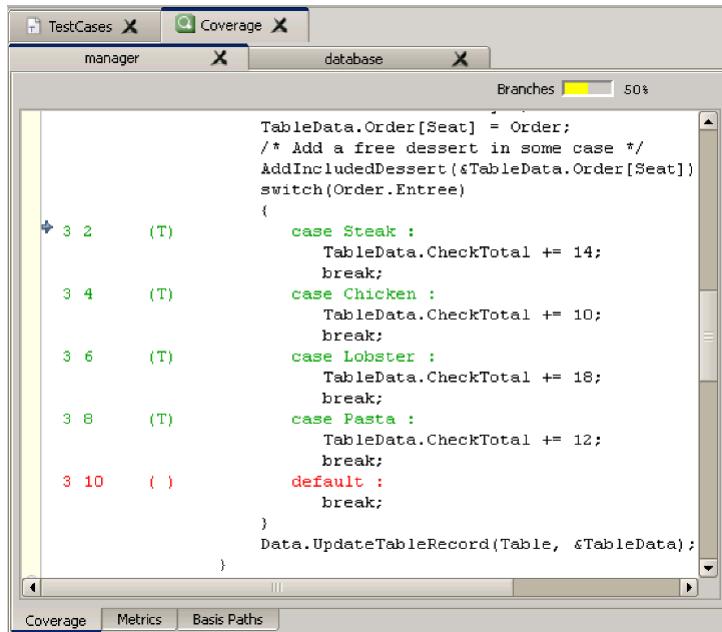


- To view the coverage data, click the green **checkbox** preceding <<COMPOUND>>.001 in the Environment View.

The Coverage Viewer opens.



12. Click the **manager** tab at the top of the Coverage Viewer to see the coverage for the unit manager.



Notice from the green coloring that by iterating over four entrées, you exercised four of the five **switch** branches. Notice from the status bar that total branch coverage was 10 of 23, or 43%. At this point, you might want to initialize a different type of coverage. In addition, you might want to try adding a test case for **clearTable** (with **Table** also set at **2**).

Exporting a Test Script

1. Click on the top level of the environment, **TUTORIAL_MULTI**, in the Environment View.
2. Select **Test => Scripting => Export Script...**, and give the test script the name **TUTORIAL_MULTI.tst**.
3. Click **Save** to create the test script.

Tutorial Summary

This tutorial built on the environment you created in the Basic Tutorial. You changed one of the dependent units from a smart stub into a UUT. You then added test cases to both units, and combined these test cases into a Compound test case, which you used to test the data flow from `manager` to `database`. When test cases are combined into a Compound test case, the data remains persistent between them.

In sum, you:

- > Built a multi UUT environment using the environment script from the Basic Tutorial as a starting point
- > Built a class constructor
- > Added a test case for the subprogram `PlaceOrder` in `manager`
- > Added a test case for the subprogram `GetTableRecord` in `database`
- > Used these simple test cases to create a compound case
- > Verified the data flow between UUTs `manager` and `database`
- > Set up range iterations for a scalar parameter and an enumeral
- > Added a test case to an existing compound case

Ada Tutorials

Introduction

The tutorials in this chapter demonstrate how to use VectorCAST/Ada to unit-test an application written in the Ada programming language.

For a comprehensive explanation of the VectorCAST/Ada features, refer to the *VectorCAST/Ada User's Guide*.

The modules you will test in the first two tutorials are components of a simple order-management application for restaurants. The source listings for this application are available in Appendix A, "Tutorial Source Code Listings" on page 311. It is recommended that you at least scan through these listings before proceeding with the tutorials.

You should run the tutorials in this chapter in the order presented.



Tip: You can stop a tutorial at any point and return later to where you left off. Each tutorial session is automatically saved. To return to where you left off, simply restart VectorCAST and the use **File => Recent Environments** to reopen the tutorial environment.

Basic Tutorial

The Basic Tutorial introduces you to all the steps necessary to build a test environment, generate test cases, produce reports, and verify code coverage.

What You Will Accomplish

In this tutorial, you will:

- > Build a test environment (which includes an executable test harness)
- > Create test cases for a unit under test (UUT)
- > Execute tests on a UUT
- > Analyze code coverage on a UUT



Note: User entries and selections are represented in this tutorial in bold. For example: "Enter **Demo**, then click **OK**".

Preparing to Run the Tutorial

Before you can run this tutorial, you need to decide on a working directory and copy into this directory a set of files from the VectorCAST installation directory. On Windows, a good choice is the default working directory, named Environments, located in the VectorCAST installation directory. On UNIX, it is often desirable to create your own working directory.

For efficiency, this tutorial assumes a working directory named vcast_tutorial located at the top level in your file tree, and a sub-directory named ADA. You can, however, locate this directory any place you want, and make the necessary translations in the text as you progress through the tutorial. Also for efficiency, this tutorial assumes you will be running the tutorial in a Windows environment, using Windows syntax for file paths and things specific to an operating system (Windows, Solaris, and UNIX). Again, you should have no problems making the necessary translations.

The files you need to copy are located in the directory in which you installed VectorCAST. On Windows, these files are located in %VECTORCAST_DIR%\Tutorial\ada by default.

In Windows, enter:

```
C:\>copy %VECTORCAST_DIR%\Tutorial\ada vcast Tutorial\ADA
```

In UNIX or Cygwin, enter:

```
$ cp $VECTORCAST_DIR/Tutorial/ada/* vcast Tutorial/ADA
```

In a Windows or shell window, change directory to the directory into which you copied the tutorial files, and then compile the Ada files using a compiler included in your PATH environment variable.

For example, to use the Green Hills Ada compiler, you would type:

```
gac types.ads manager.ads database.ads whitebox.ads
types.ads:
database.ads:
manager.ads:
whitebox.ads:
gac database.adb manager.adb whitebox.adb
database.adb:
manager.adb:
whitebox.adb:
```

To use the GNAT Ada compiler, you would type:

```
gcc -c manager.adb database.adb whitebox.adb types.ads
ls
database.adb      manager.adb      manager_driver.adb      whitebox.adb
database.ads      manager.ads      types.ads          whitebox.ads
database.ali      manager.ali      types.ali          whitebox.ali
database.o        manager.o       types.o           whitebox.o
```

Building an Executable Test Environment

In this section, you will use the VectorCAST/Ada tool to create an executable test environment. The sample UUT you will use is named **manager**.

You will:

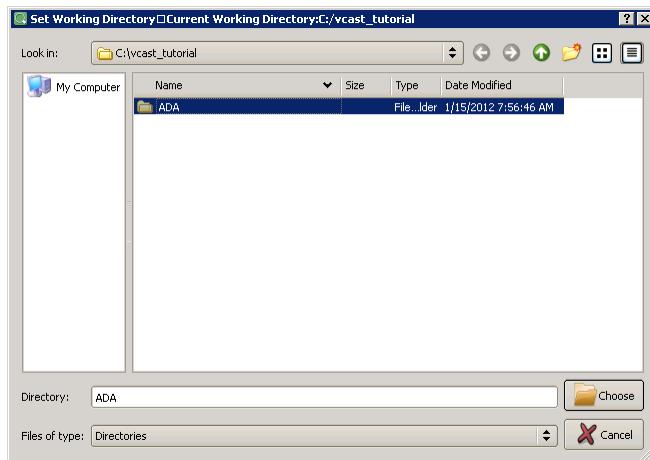
- > Specify a working directory
- > Specify a compiler
- > Name the environment to be created
- > Specify the source files to be included in the environment
- > Designate the UUT and any stubs (as well as the method of stubbing)
- > View a summary of your specifications
- > Build the test environment

Specifying a Working Directory

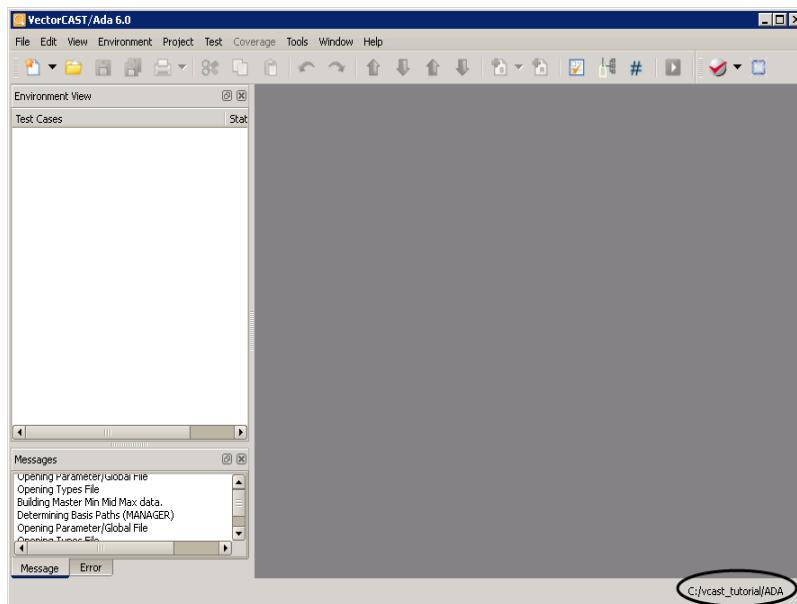
The working directory contains the files you will use to build your test harness. For the purposes of this tutorial, the working directory is **vcast_tutorial\ADA**

For details on starting VectorCAST, refer to "Introduction" on page 7.

1. With the VectorCAST main window open, select **File => Set Working Directory**.
A browsing dialog appears.
2. Navigate to **C:\vcast_tutorial\ADA** ; click **Choose**.



The location and name of the working directory you specified appear at the lower right-hand corner of the VectorCAST main window:



Before you Begin

If you plan to use a version of the GNAT compiler prior to version 5, then you need to turn off a Builder option. If you are using GNAT 5 or greater, or a different compiler altogether, then you do not need to make any changes.

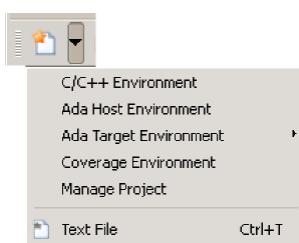
If you are using GNAT 4 or earlier, follow these steps:

1. Choose **File => Set Product Mode => Ada Host Environment**
2. Choose **Tools => Options**.
3. In the Option dialog, click the **Builder** tab.
4. Uncheck the option “GNAT version 5 or greater”
5. Click **OK**.

Specifying the type of environment to build

1. Click the **New** button  on the toolbar.

If more than one VectorCAST product is licensed, a drop-down menu appears listing the available environment types:



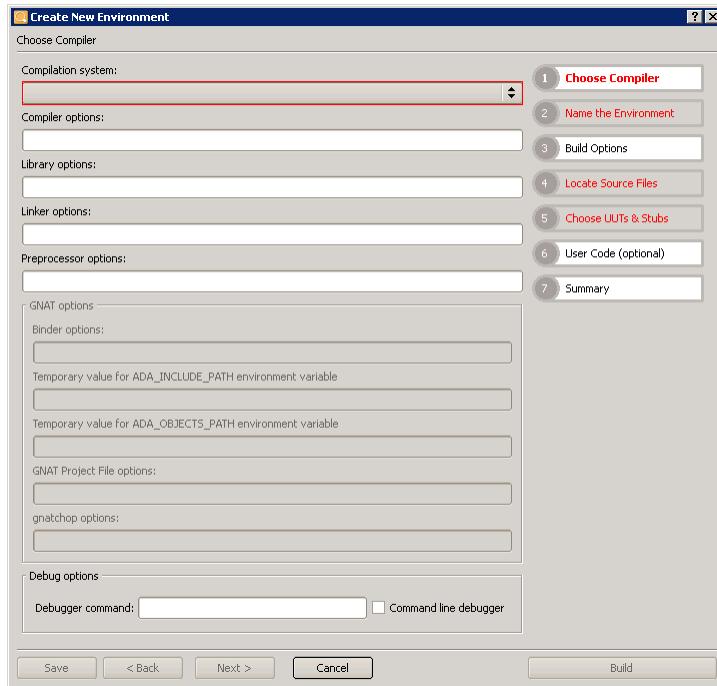
If the only type of environment licensed is Ada, the Create New Environment wizard appears when you click the New button.



Tip: If you are planning to use VectorCAST with a target compiler, it is recommended you first go through the tutorials using a Host version of your compiler. Once you are familiar with the basic functionality of VectorCAST, refer to the VectorCAST/RSP User's Guide for information specific to your target.

2. Select **Ada Host Environment**.

The Choose Compiler page of the Create New Environment wizard appears:

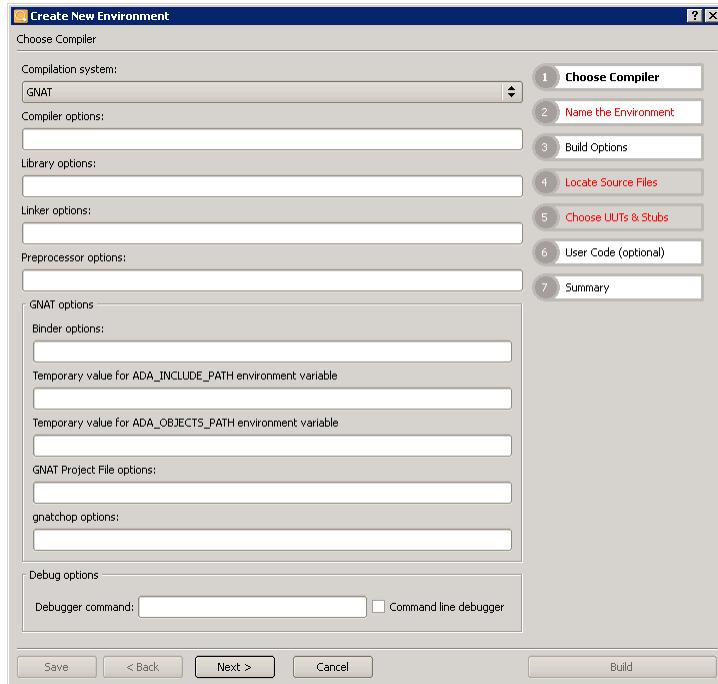


Specifying a compiler

You use the Choose Compiler page to specify a compiler. The compiler you specify must be included in your PATH environment variable.

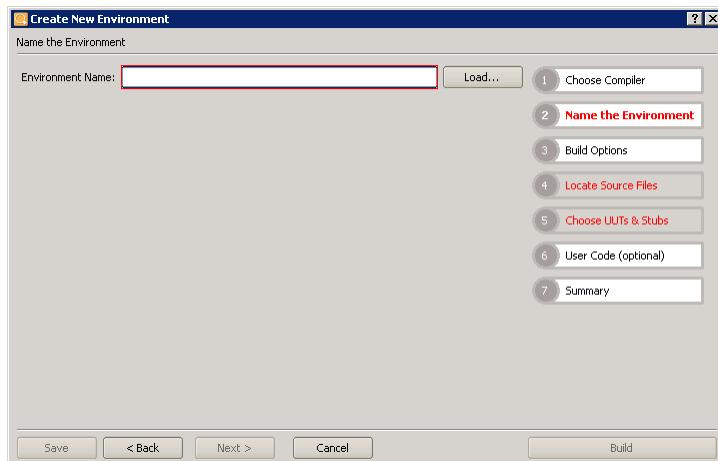
1. To access a list of Ada compilers supported by VectorCAST, click the down-arrow at the far right of the **Compilation system** field (bordered in red).
2. Select an Ada compiler from the drop-down menu that appears.

When you select a compiler, the red border disappears from the Compilation system field:



3. Click **Next**.

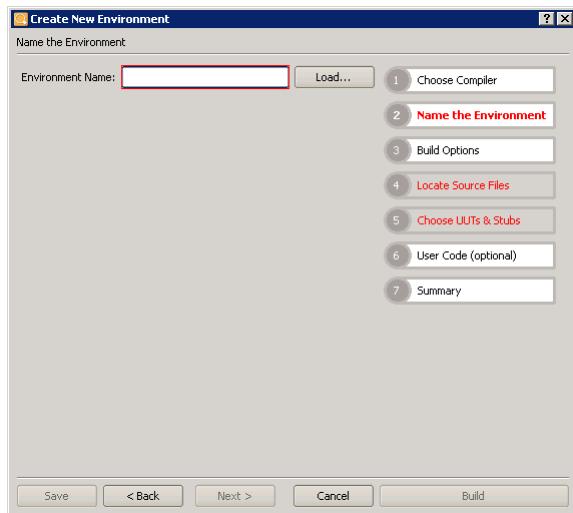
The Name the Environment page appears:



Naming your test environment

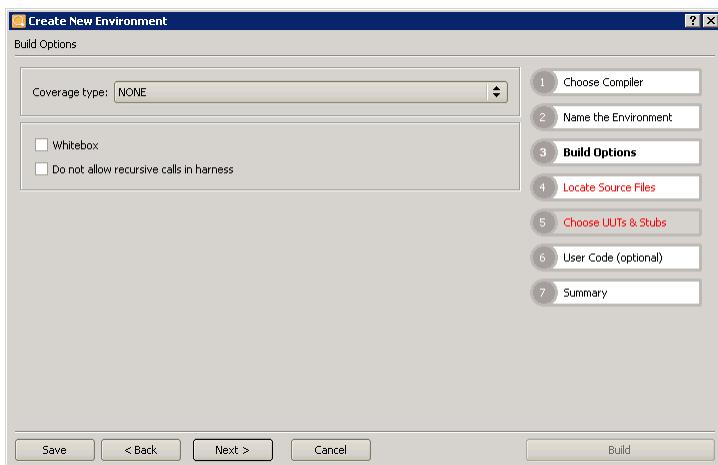
For the purposes of this tutorial, you will name your environment 'tutorial'.

1. Enter **tutorial** into the Environment Name field.
VectorCAST echoes the letters in uppercase:



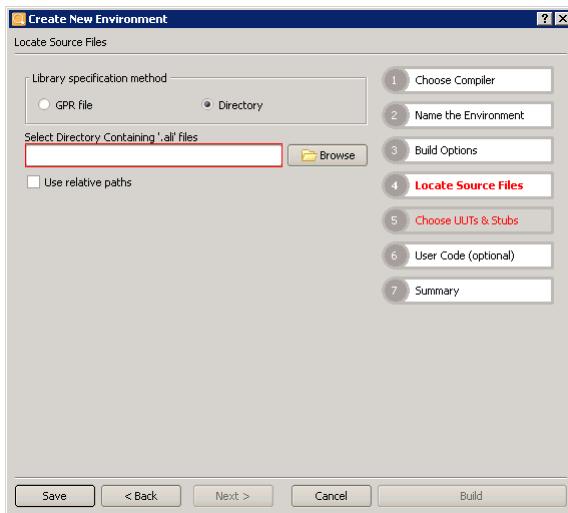
2. Click **Next**.

The Build Options page appears:



3. We'll be initializing coverage later in the tutorial; for now, just click **Next**.

The Locate Source Files page opens:

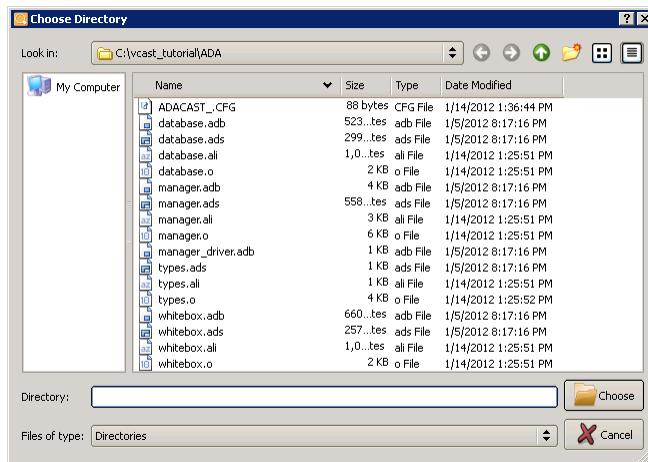


Note: The text on this page depends on the compiler you selected.

Specifying the source files to be tested

To specify the directory in which your Parent Library resides:

1. Click the **Browse** button  on the **Locate Source Files** page to display the Choose Directory dialog:

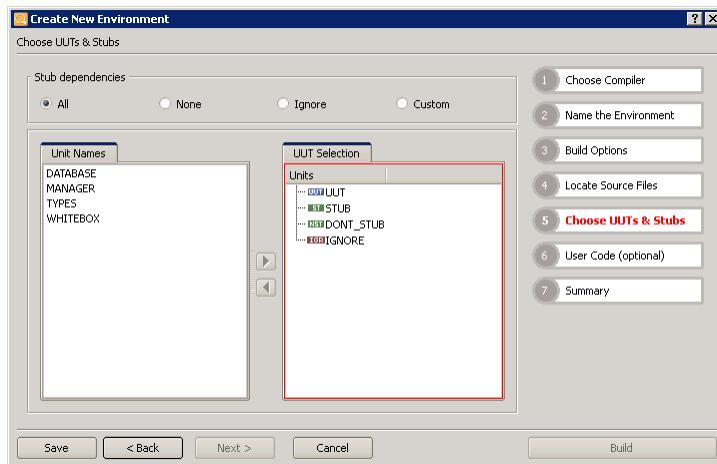


In this tutorial, you are building your environment in the same location in which the source files to be tested are located. In practice, you might need to navigate to a different directory.

2. Click **Choose**.

3. Click **Next**.

The Choose UUTs & Stubs page appears:

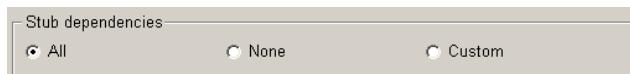


Designating UUTs and stubs

You use the Choose UUTs & Stubs page to designate units as a UUT or stub and the method of stubbing.

The pane under Unit Names lists the units that VectorCAST found in the specified source directory. You can select one or more of these units to be UUTs. For this tutorial, you will select only MANAGER.

1. Click MANAGER under Unit Names, and then click the move-right button; or simply double-click MANAGER. The name MANAGER appears under .**UUT** UUT in the Units column:
Three options are available under Stub dependencies for designating stubbing: All, None, or Custom:



As indicated, VectorCAST defaults to stubbing all dependent units. You can override this setting by selecting None or Custom or by telling VectorCAST to ignore all dependent units. The Custom option allows you to select individual subprograms for stubbing.

See *VectorCAST/Ada User's Guide* for information on using the **ST** STUB and **IGR** IGNORE nodes in conjunction with the Stub dependencies settings.

For the purposes of this tutorial you will accept the default setting (All).

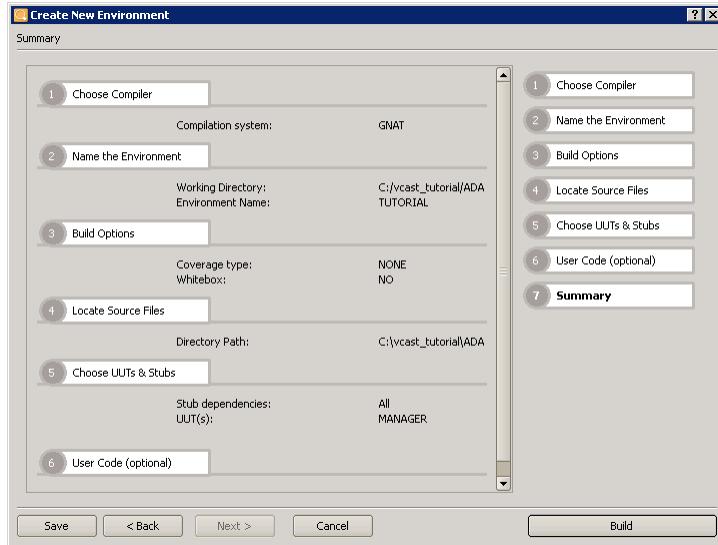
2. Click **Next**

The User Code appears. The User Code page is optional and not needed for this tutorial.

Viewing a summary of your test-environment specifications

1. Click **Next**.

The Summary page appears:



This page allows you to check your test-environment specifications before you generate the test harness.

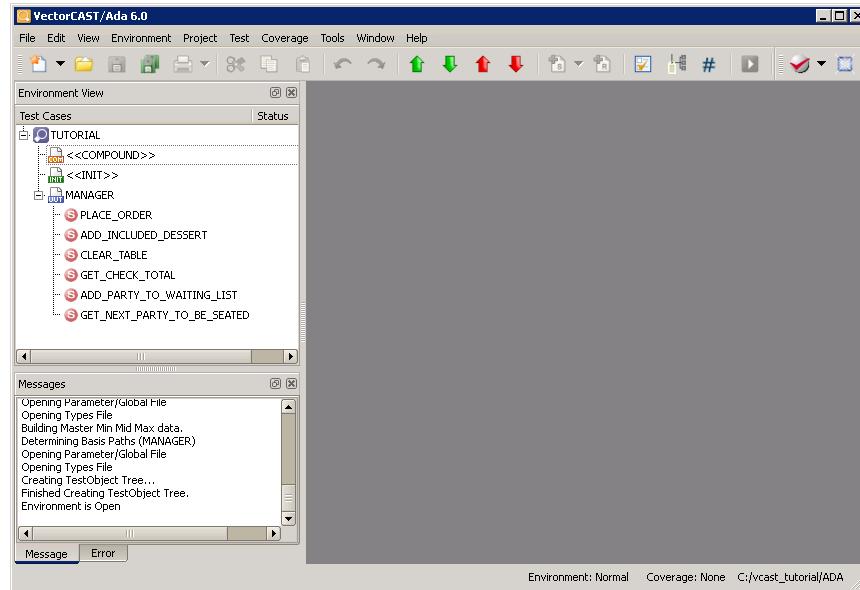
Generating an executable test harness

To compile your specifications into an executable test harness:

1. Click the **Build** button to compile your test-environment specifications you generate the test harness.

As an environment is built, VectorCAST displays a series of messages in the Message window. VectorCAST is parsing the UUT MANAGER, creating a stub for DATABASE, and compiling and linking the test harness.

When processing has completed, VectorCAST displays the main window:



VectorCAST has created an environment file named **TUTORIAL.vce** and a subdirectory named **TUTORIAL**.

The Environment View shows that the current environment (TUTORIAL) has one UUT, named **MANAGER**, which has six subprograms:

At this point, VectorCAST has built the complete test harness necessary to test the **MANAGER** unit, including:

- > A test directory with all test-harness components compiled and linked.
- > The test driver source code that can invoke all subprograms in the file (**MANAGER**).
- > The stub source code for the dependent unit (**DATABASE**).

Note what was accomplished during environment creation:

- > The unit under test was parsed. The dependent unit did not have to be parsed, because its prototype was used to determine the parameter profile.
- > The data-driven driver and stub source code were generated.
- > The test code generated by VectorCAST was compiled and linked with the UUT to form an executable test harness.

The harness that VectorCAST builds is *data driven*. This means that data sets can be built to stimulate the different logic paths of the unit under test without ever having to re-compile the harness. You are now ready to build test cases for the subprograms in the unit under test (MANAGER).

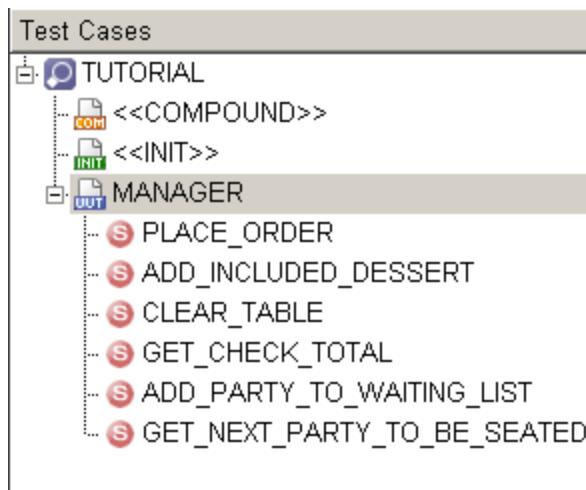
Building Test Cases

In this section, you will build a test case to verify the operation of the subprogram **PLACE_ORDER**.

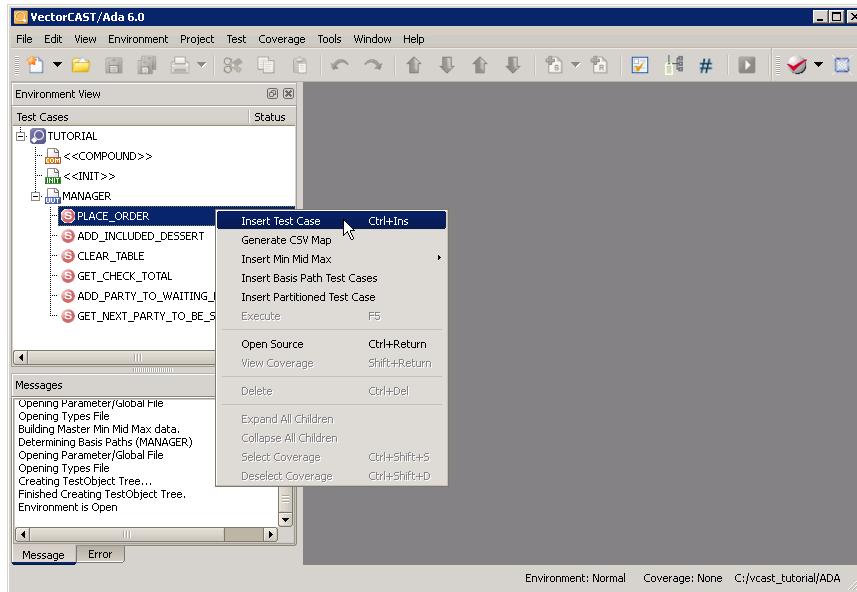
The process of building test cases with VectorCAST is very simple. All the information is available by means of a point-and-click interface. The idea is to create a set of data that will stimulate the unit under test and cause it to execute a particular logic path. In its simplest form, a test case consists of values for the formal parameters of a single subprogram within the unit under test. A more complex test case can also contain values to be returned by a stubbed dependent, or values to be set into global data objects.

Test cases are constructed to stimulate individual subprograms. Each subprogram within a unit will have its own series of test cases defined. The first step is to select a subprogram in the Environment View and create a test case for it.

1. If a list of subprograms is not displayed under **manager** in the Environment View, click the symbol preceding its name to display the list of subprograms.



2. In the Environment view, PLACE_ORDER.001 appears as a test-case item under PLACE_ORDER.



USER_GLOBALS_VCAST	A collection of global objects for use as a workspace
MANAGER	The unit under test
PLACE_ORDER	The subprogram to be tested
<<raise>>	Exception Handler
TABLE	Subprogram of type Integer
SEAT	Subprogram of type Integer
ORDER	Subprogram of type Record
Stubbed Subprograms	A collapsed tree of stubbed subprograms from dependent unit(s)

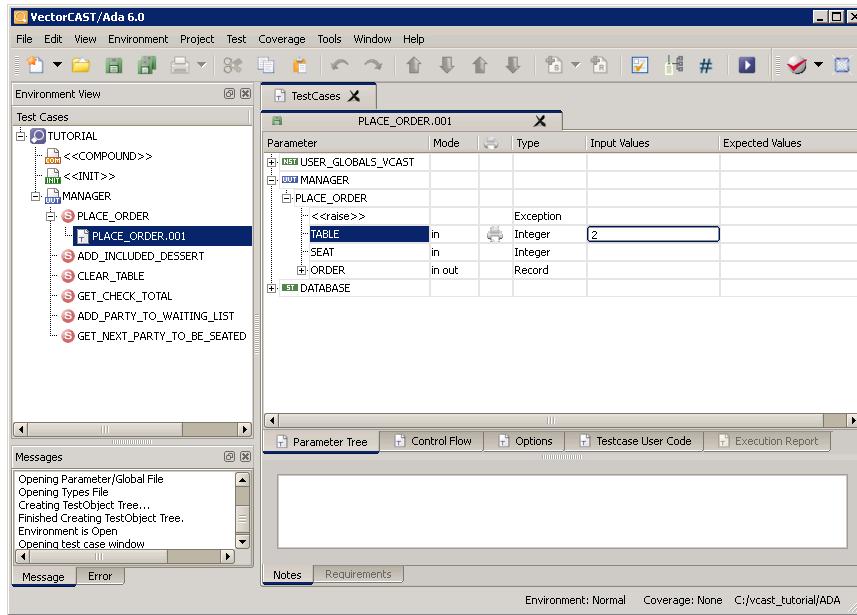


Note: You can change the default name of a test case by right-clicking its name and selecting **Rename** from the popup menu.

The following test case parameter tree appears in the Test Case Editor:

A tree item having branches is preceded by a plus symbol \oplus . To expand a tree item, simply click the \oplus symbol.

3. For the parameter **Table**, enter **2** into **Input Values** field:

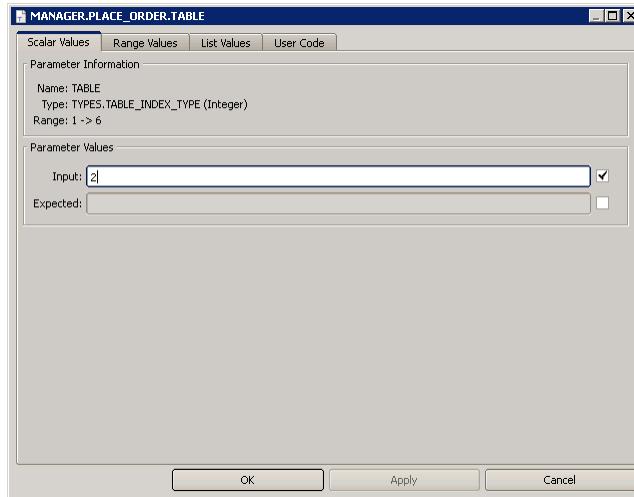


4. Press **Enter.**

Table 2 is now the focus of your test case.

5. Double-click **Table.**

The parameter dialog box appears:



Notice that you can use this dialog to set simple scalar values, and also to build test cases for a range of values or a list of values, or with user code.

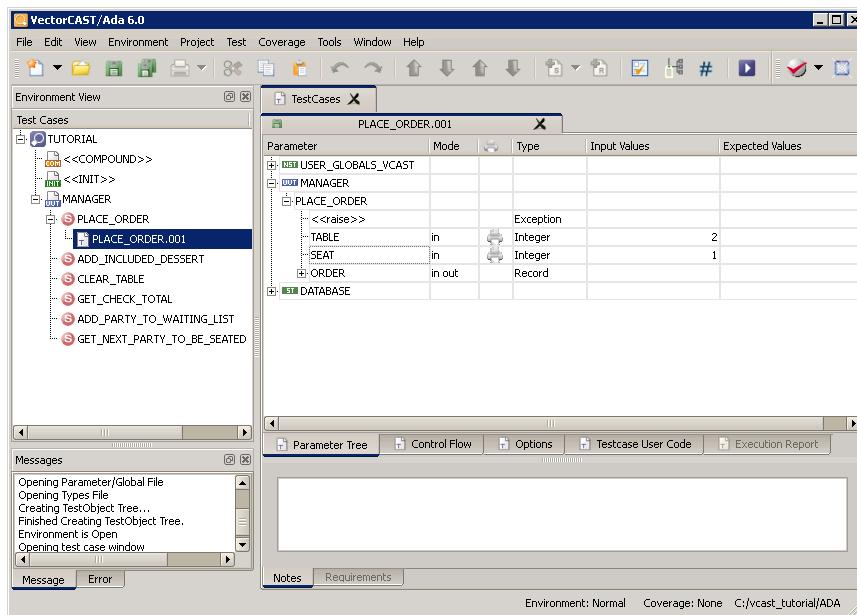
6. Click **Cancel to dismiss this dialog.**

You will use it in the next tutorial.



Note: When you create a test case, the only subprogram displayed for the UUT is the subprogram that this test case has been created for, in this case, PLACE_ORDER.

7. For this tutorial, assume now that someone is occupying seat 1 at table 2.
Enter **1** into the **Input Values** field for **Seat**, and then press **Enter**.



You will use this same technique to set values for all parameters. Although you can access all parameters of visible subprograms of the unit under test, and of any stubbed unit, it is only necessary to select those parameters that are relevant to the test you are running.

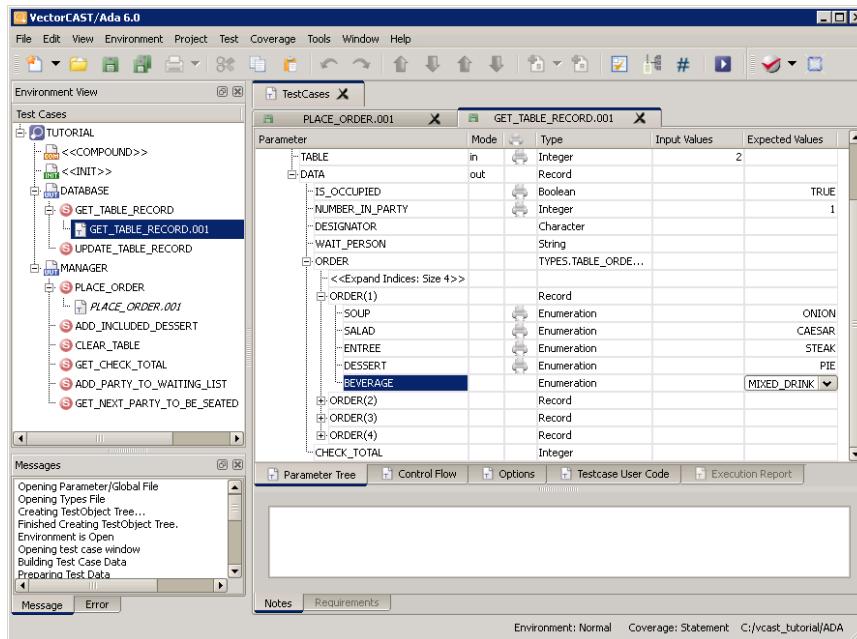
Non-scalar parameters (arrays and structures) are accessed using the same tree-expansion technique that is used for units and subprograms. Parameter **Order** is a structure with five fields. Each field is an enumeration type.

At this point, you could create separate test cases for the other seats at table 2 (as well as for the other tables in the restaurant); however, for the purposes of this basic tutorial, one test case (one order at one seat) is sufficient.

You have someone sitting in seat 1 at table 2; it's time now to take an order.

8. Expand **Order** into its fields by clicking .
- Five order fields appear: Soup, Salad, Entree, Dessert, and Beverage. (The specifications for these order fields are included in "Tutorial Source Code Listings" on page 311.)
9. For each order field, click in the **Input Values** column and select a value from the drop-down menu that appears.
- For the purposes of this example, select the following values:
- | | |
|-------|---------------|
| Soup | ONION |
| Salad | CAESAR |

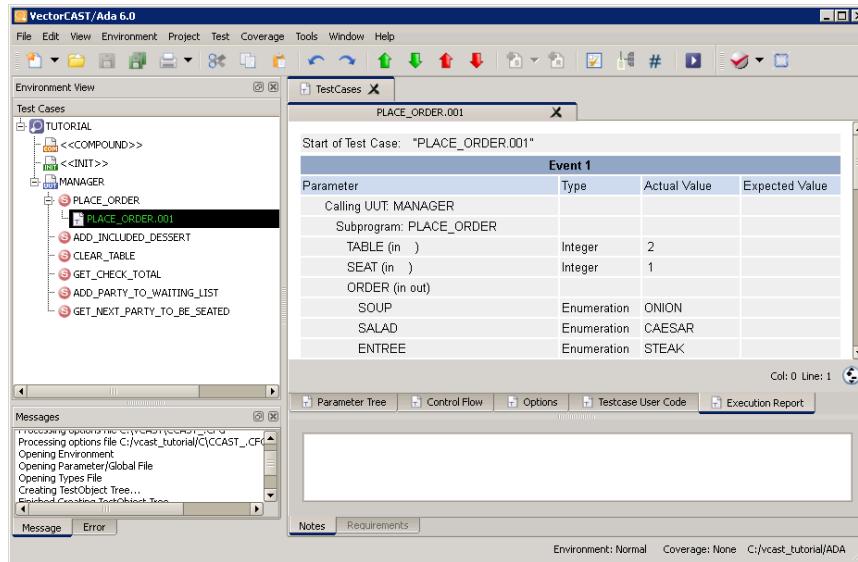
Entree **STEAK**
 Dessert (leave unspecified)
 Beverage **MIXED_DRINK**



Note: To clear a field, select the blank item at the top of the associated drop-down menu.

10. Save test case PLACE_ORDER.001: Either select **File => Save** from the main-menu bar or click the Save  button on the toolbar.
11. To access a test-case data listing, select **Test => View => Test Case Data**. The Test Case Data Report for your test case appears in the Report Viewer.
12. In the Table of Contents section of this report, click **Test Case Data**.

The report jumps to the Test Case Data section.



13. Close the report by clicking the X on the report's tab.

You have now built a test case that is ready to be executed by VectorCAST, without any compiling of data or writing of test code. The data that was set up for this case corresponds to the formal parameter list for subprogram **PLACE_ORDER**. You set data for scalar parameters **TABLE** and **SEAT** and for structure parameter **ORDER**.

At this point you could execute your test case as is, or you could add expected results for comparing against actual results. Because the latter course is generally the more valuable, you will now add expected results to your test case.

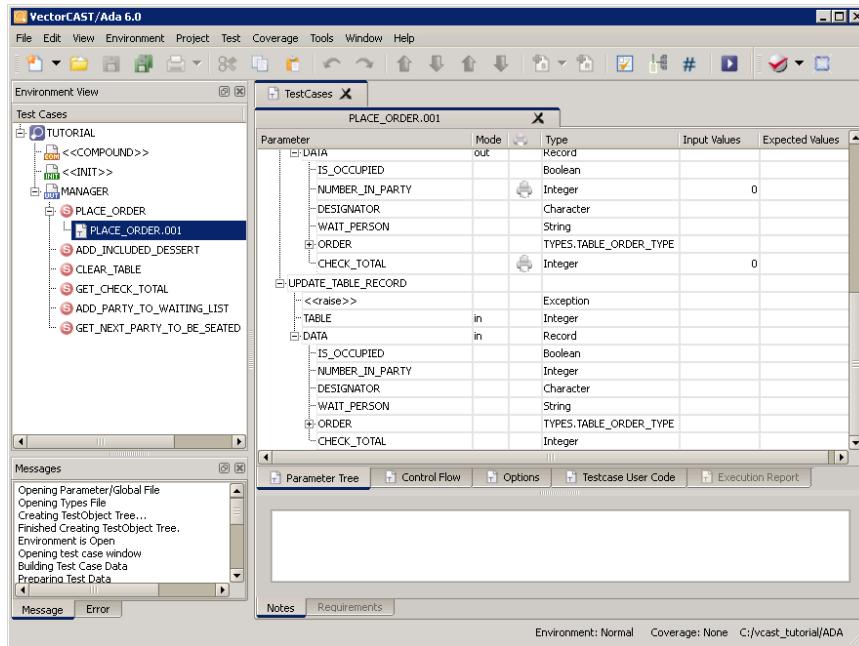
Controlling Values Returned from Stubs

VectorCAST provides you with a simple way to define expected results for each test case. You can add expected results at the time you set input values or at a later time. As a test case is executed, the expected results are compared against actual results to determine whether the test case passed or failed.

In this section, you will add expected results to the test case you created in the previous section:

1. Click the \oplus preceding **DATABASE**; then click the \oplus preceding **GET_TABLE_RECORD**. Then click the \oplus preceding **DATA**.

DATA expands into its fields:



Note: If you encounter any problems, refer to the section titled *Troubleshooting*.

You can now enter expected values for NUMBER_IN_PARTY and CHECK_TOTAL. Before you do this, however, you need to initialize the data returned by GET_TABLE_RECORD.

2. To initialize the data returned by **GET_TABLE_RECORD**, enter **0** into the Input Values fields for **NUMBER_IN_PARTY** and **CHECK_TOTAL**

ST	DATABASE	
	GET_TABLE_RECORD	
	... <<raise>>	Exception
	TABLE	in Integer
	DATA	out Record
	IS_OCCUPIED	Boolean
	NUMBER_IN_PARTY	Integer 0
	DESIGNATOR	Character
	WAIT_PERSON	String
	ORDER	TYPES.TABLE_ORDER_TYPE
	CHECK_TOTAL	Integer 0

You are now ready to enter expected values for comparing against the actual values returned by **UPDATE_TABLE_RECORD**.



Tip: You might want to make more room by closing up the Parameter trees for both **GET_**



manager.

Adding Expected Results to the Test Case

VectorCAST provides you with a simple way to define expected results for each test case. You can add expected results at the time you set input values or at a later time. As a test case is executed, the expected results are compared against actual results to determine whether the test case passed or failed.

In this section, you will add expected results to the test case you created in the previous section:

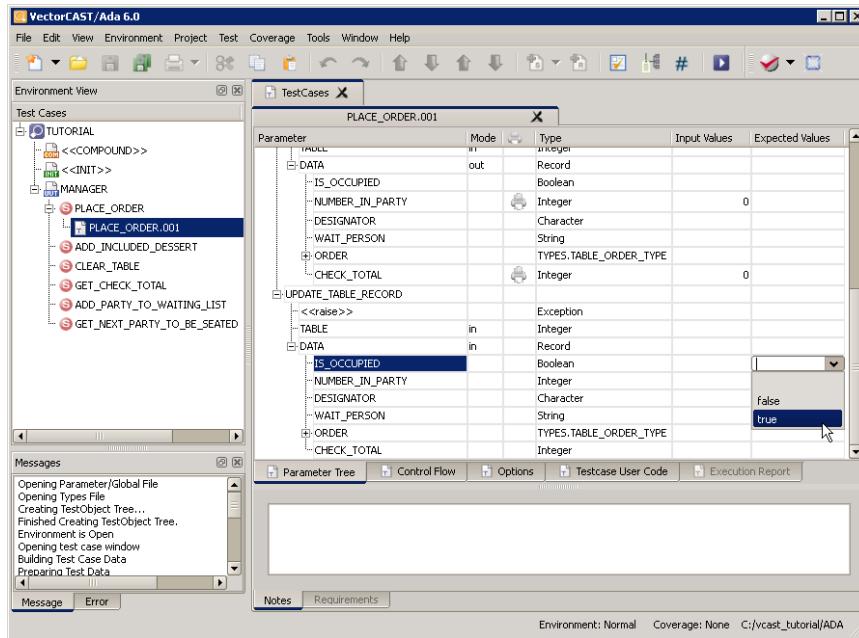
1. Click the preceding **UPDATE_TABLE_RECORD**, then Click the preceding **DATA**. DATA expands into its fields:

Parameter	Mode	Type	Input Values	Expected Values
IS_OCCUPIED	out	Boolean		
NUMBER_IN_PARTY	in	Integer	0	
DESIGNATOR	in	Character		
WAIT_PERSON	in	String		
ORDER	in	TYPES.TABLE_ORDER_TYPE		
CHECK_TOTAL	in	Integer	0	
DATA	in	Record		
IS_OCCUPIED	in	Boolean		
NUMBER_IN_PARTY	in	Integer		
DESIGNATOR	in	Character		
WAIT_PERSON	in	String		
ORDER	in	TYPES.TABLE_ORDER_TYPE		
CHECK_TOTAL	in	Integer		

In the **Expected Values** column, enter expected results as follows:

IS_OCCUPIED: true

NUMBER_IN_PARTY: 1



Note: Be sure you enter your values into the **Expected Values** column, and not into the **Input Values** column.

2. In addition, enter **12..16** as the range of expected results for **CHECK_TOTAL**.

UPDATE_TABLE_RECORD					
-<<raise>>		Exception			
TABLE	in	Integer			
DATA	in	Record			
-IS_OCCUPIED		Boolean	TRUE		
--NUMBER_IN_PARTY		Integer	1		
-DESIGNATOR		Character			
-WAIT_PERSON		String			
ORDER		TYPES.TABLE_ORDER_TYPE			
CHECK_TOTAL		Integer	12..16		

3. Expand **ORDER**

ORDER is an array whose index corresponds to the seat specified in the call to **PLACE_ORDER**. Because we specified seat 1, we need to expand **ORDER (1)**.

4. In the Input Values column, enter **1** next to <<Expand Indices: Size 4>>.

UPDATE_TABLE_RECORD					
<<raise>>			Exception		
TABLE	in		Integer		
DATA	in		Record		
IS_OCCUPIED			Boolean	TRUE	
NUMBER_IN_PARTY			Integer	1	
DESIGNATOR			Character		
WAIT_PERSON			String		
ORDER			TYPES.TABLE_ORDER_TYPE		
<<Expand Indices: Size 4>>				1	
CHECK_TOTAL			Integer		12..16

5. Press **Enter**.

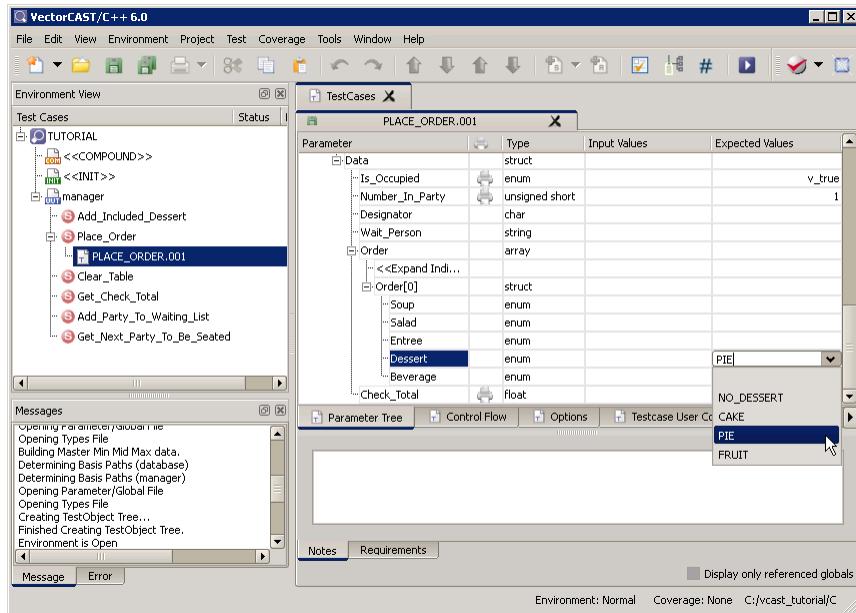
UPDATE_TABLE_RECORD					
<<raise>>			Exception		
TABLE	in		Integer		
DATA	in		Record		
IS_OCCUPIED			Boolean	TRUE	
NUMBER_IN_PARTY			Integer	1	
DESIGNATOR			Character		
WAIT_PERSON			String		
ORDER			TYPES.TABLE_ORDER_TYPE		
<<Expand Indices: Size 4>>					
ORDER(1)			Record		
CHECK_TOTAL			Integer		12..16

The array element (1) in the array ORDER is expanded in the Parameter Tree, and the “1” is removed.

6. Expand the tree for ORDER (1) .

UPDATE_TABLE_RECORD					
<<raise>>			Exception		
TABLE	in		Integer		
DATA	in		Record		
IS_OCCUPIED			Boolean	TRUE	
NUMBER_IN_PARTY			Integer	1	
DESIGNATOR			Character		
WAIT_PERSON			String		
ORDER			TYPES.TABLE_ORDER_TYPE		
<<Expand Indices: Size 4>>					
ORDER(1)			Record		
CHECK_TOTAL			Integer		12..16

7. For the Expected Value for Dessert, enter **PIE**. We expect Pie for dessert because PLACE_ORDER calls ADD_INCLUDED_DESSERT, which sets the DESSERT to PIE.



You now have a test case defined with input data and expected results.

The values entered into the Expected Values column for a stub are verified as they are passed into the stub; the values entered into the Input Values column are passed out of the stub and back into the test harness.

You must now save your data in order to make them a part of your test case.

When a test case has been modified and needs to be saved, a green icon () appears to the left of its name on a tab above the Test Case Editor

8. To save your modifications, click the **Save** button  on the toolbar.
9. To view the saved data, select **Test => View => Test Case Data**, then click the link **Test Case Data**.

Table 1. Test Case Data

Test Case: "PLACE_ORDER.001"

File Name: "C-000001.DAT"

Input Test Data	
UUT: MANAGER	
Subprogram: PLACE_ORDER	
TABLE (in)	2
SEAT (in)	1
ORDER (in out)	
SOUP	ONION
SALAD	CAESAR
ENTREE	STEAK
BEVERAGE	MIXED_DRINK
Stubbed Unit: DATABASE	

Subprogram: GET_TABLE_RECORD	
DATA(out)	
NUMBER_IN_PARTY	0
CHECK_TOTAL	0
Expected Test Data	
Stubbed Unit: DATABASE	
Subprogram: UPDATE_TABLE_RECORD	
DATA(in)	
IS_OCCUPIED	TRUE
NUMBER_IN_PARTY	1
ORDER	
(1)	
DESSERT	PIE
CHECK_TOTAL	BETWEEN:12.0 AND:16.0
Test Case / Parameter Input User Code	
Test Case / Parameter Expected User Code	



If your test case Data report does not show any data in the section labeled Expected Test Data, refer to the section titled Troubleshooting at the end of this tutorial.

- To close this report, click the X on the Test Case Data tab. To close all reports, click on the X on the Reports tab.

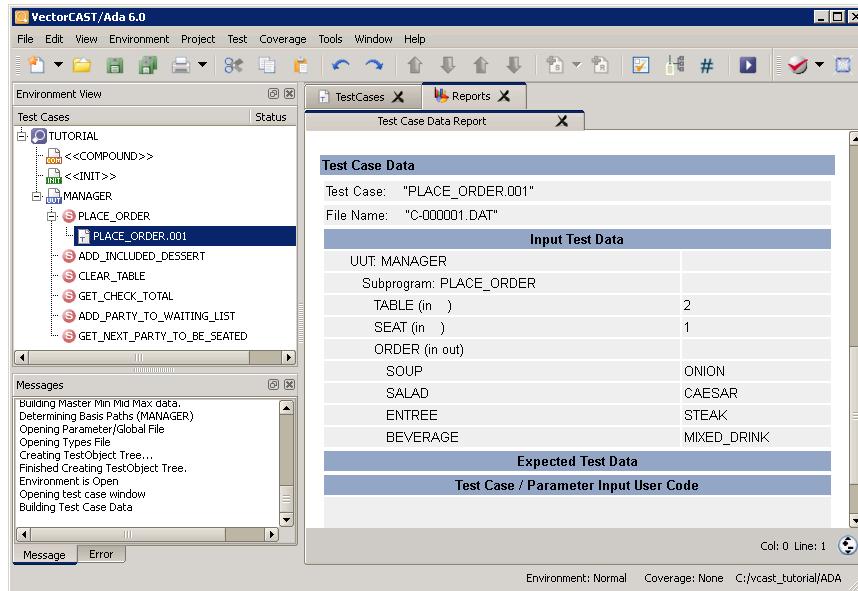
You have now created a test case named PLACE_ORDER.001 that includes expected results. You are now ready to execute it.

Executing Test Cases

In this section, you will execute the test case you created in the previous sections.

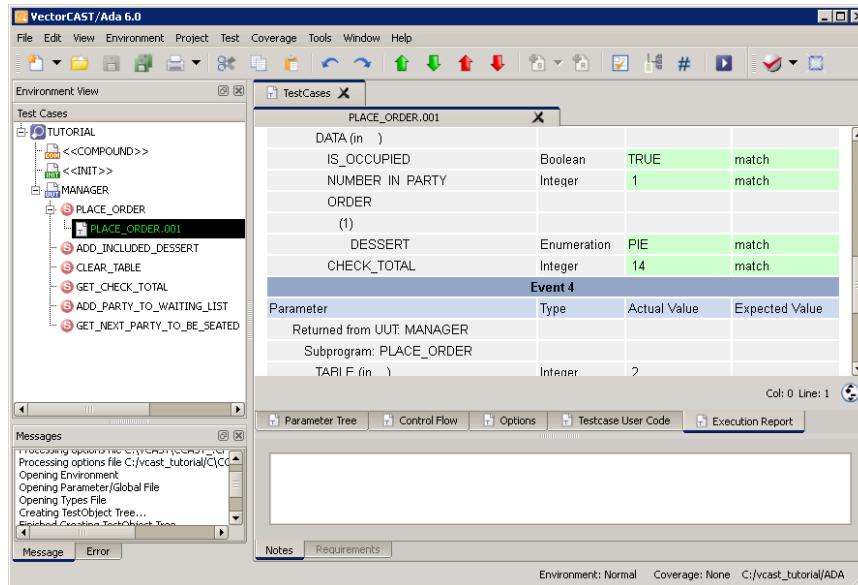
- To execute your test case, click **PLACE_ORDER.001** in the Environment View, then click the **Execute** button on the toolbar.

When processing completes, an execution report appears in the Report Viewer.



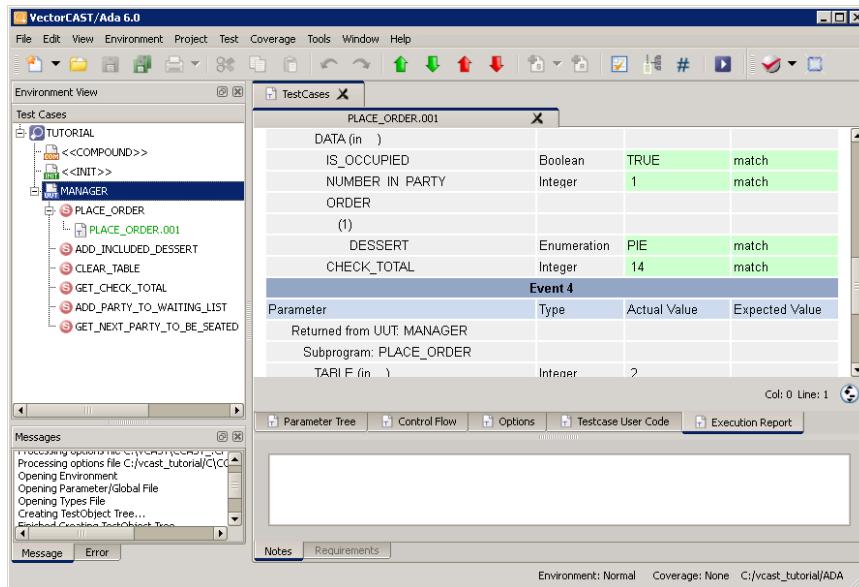
- Click the green down-arrow on the toolbar, or scroll down the report until you come to a group of table cells shaded in green.

The green shading and the term **match** here indicate that the expected values you specified were matched by actual values. Your test passed!



Note that **PLACE_ORDER.001** in the Environment View is now green.

This color-marking signifies that the most-recent execution of test case PLACE_ORDER.001 passed. In addition, the Status field displays PASS in green, while the Date field displays the hour-minute-second of the execution.



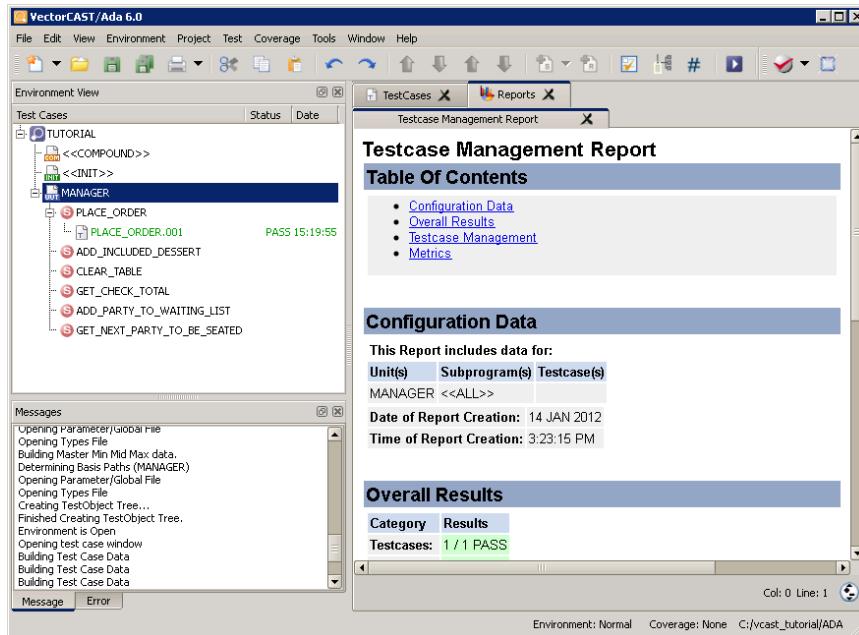
Generating and Managing Test Reports

In this section, you will view the Test Case Management Report for the recent execution of test case PLACE_ORDER.001.

A Test Case Management Report includes:

- > The results of all tests conducted with this environment
 - > The number of test cases that passed
 - > The number of comparisons made that passed
 - > The pass-fail status of all test cases
 - > Metrics data, including cyclomatic complexity and, if code coverage was enabled, the coverage achieved
1. To view the Test Case Management Report for your environment, select **Test => View => Test Case Management Report** from the main-menu bar.

The Test Case Management Report for environment TUTORIAL appears in the Report Viewer:



Note: You can customize the appearance of this or any other VectorCAST report. To customize a report, select **Tools => Options**; click the **Report** tab; then click the **Format** sub-tab.

2. Use the scrollbar to move through the report.
3. To print your Test Case Management Report, click the **Print** button on the toolbar  , or select **File => Print** from the main-menu bar.
4. To save your report into a file, select **File => Save As**.

When the **Save As** dialog box opens, enter **tutorial_report.html** for the file name, then click **Save**.



Note: You can open a saved report with any HTML browser (Internet Explorer, Mozilla, Netscape, etc.).

Analyzing Code Coverage

Code coverage analysis enables you to see which parts of your application are being stimulated by your test cases. Once coverage is initialized for an environment, you can choose to run your VectorCAST test cases with either the coverage-instrumented test harness or the uninstrumented harness. Coverage enables you to view a code coverage report for individual test cases and an aggregate report showing coverage resulting from any subset of test cases.

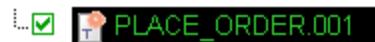
The Units Under Test must be instrumented for the kind of coverage you want: Statement, Branch, MC/DC, DO-178B Level A, DO-178B Level B, or DO-178B Level C. In this section, you will instrument the source files in your environment for statement coverage:

- To instrument the source files in your environment for statement coverage, select **Coverage => Initialize => Statement**.

The status bar on the bottom right of the VectorCAST main window now indicates that statement coverage is active:



- To execute an existing test case and generate a report, click **PLACE_ORDER.001** in the Environment View, then click the execute button (▶).
- When coverage is enabled, VectorCAST keeps track of the lines or branches in the source code that are traversed during execution. You can access a report on the results by way of clicking the checkbox next to the test case name in Environment View.
- To access the coverage report for your test case, click the **green checkbox** to the left of **PLACE_ORDER.001**.



An annotated version of the **MANAGER** source code appears in the Coverage Viewer:

A screenshot of the VectorCAST Ada 6.0 Coverage Viewer. The left pane shows the "Test Cases" view with "PLACE_ORDER.001" selected. The right pane displays the "MANAGER" package body. The code is annotated with coverage information: green numbers indicate exercised statements, red numbers indicate unexecuted statements, and black numbers indicate continuation of previous statements. An asterisk (*) marks a covered statement, and a blue arrow points to the current line.

```

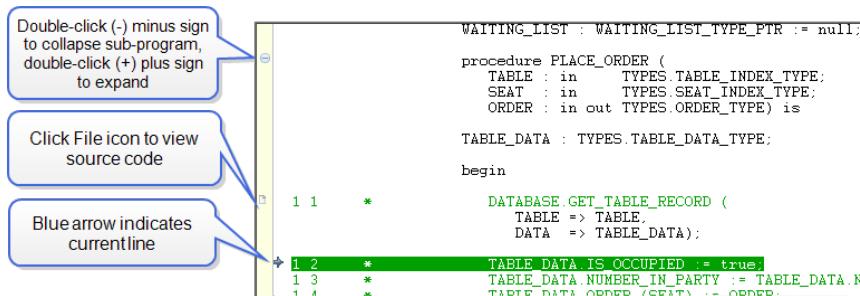
package body MANAGER is
    MAX_NAME_LENGTH : constant := 20;
    subtype WAITING_LIST_NAME_TYPE is string;
    type WAITING_LIST_TYPE;
    type WAITING_LIST_TYPE_PTR is access WAITING_LIST_TYPE;
    type WAITING_LIST_TYPE is
        record
            NAME : WAITING_LIST_NAME_TYPE := '';
            NEXT : WAITING_LIST_TYPE_PTR := null;
        end record;
    WAITING_LIST : WAITING_LIST_TYPE_PTR := null;
    procedure PLACE_ORDER (
        TABLE : in     TYPES.TABLE_INDEX_TYPE;
        SEAT  : in     TYPES.SEAT_INDEX_TYPE;
        ORDER : in out TYPES.ORDER_TYPE) is
        TABLE_DATA : TYPES.TABLE_DATA_TYPE;
    begin
        1 1      *          DATABASE.GET_TABLE_RECORD (
            TABLE => TABLE,
            DATA  => TABLE_DATA);
    end;
end;
  
```

Each set of repeating numbers in the first column marks a subprogram within **MANAGER**. The ascending numbers in the second column mark the executable statements within a subprogram. The statements in green have been exercised; the statements in red have not been exercised; the statements in black cannot be exercised, or are the continuation of previous statements.

An asterisk (*) indicates a covered statement. The blue arrow marks the current line. You can

move the current line  by means of the up and down arrows on your keyboard, or by clicking on a line.

Note the icons on the left-hand margin of the Coverage Viewer:



```

WAITING_LIST : WAITING_LIST_TYPE_PTR := null;
procedure PLACE_ORDER (
    TABLE : in      TYPES.TABLE_INDEX_TYPE;
    SEAT : in      TYPES.SEAT_INDEX_TYPE;
    ORDER : in out TYPES.ORDER_TYPE) is
    TABLE_DATA : TYPES.TABLE_DATA_TYPE;
begin
    DATABASE.GET_TABLE_RECORD (
        TABLE => TABLE,
        DATA => TABLE_DATA);
    TABLE_DATA.IS_OCCUPIED := true;
    TABLE_DATA.NUMBER_IN_PARTY := TABLE_DATA.N
    TABLE_DATA.COVERED := true;
end;

```

A circled minus-sign  means the associated subprogram is fully expanded in the Coverage Viewer, and can be collapsed. To collapse a subprogram (to its first line), double-click either the circle or the first line of the subprogram.

Clicking the file symbol gives you immediate access to the associated source code.

The status bar (upper right) tells you the number and percentage of statements exercised.

4. Hover your mouse over **Statements** on the status bar.

A tool tip shows the number of lines covered: 10 of 41 (24%).



Note: VectorCAST's animation feature allows you to view the step-by-step coverage of a test case. This feature is described in the user guide accompanying your version of VectorCAST.

The Metrics tab at the bottom of the Coverage Viewer displays a tabular summary of the coverage achieved, on a per subprogram basis.

5. Click the **Metrics** tab:

Subprogram Name	Complexity V(g)	Statements
PLACE_ORDER	5	8 / 12
ADD_INCLUDED_DESSERT	3	2 / 3
CLEAR_TABLE	2	0 / 7
GET_CHECK_TOTAL	1	0 / 2
ADD_PARTY_TO_WAITING_LIST	2	0 / 5
GET_NEXT_PARTY_TO_BE_SEATED	6	0 / 12
TOTAL	19	10 / 41

This table tells you that subprogram **PLACE_ORDER** consists of 12 executable statements; that 8 of these statements were exercised during the selected test; and that the complexity of the subprogram is 5, meaning there are five distinct execution paths.

6. Click the **Coverage** tab to return to the previous view.

For each green line, you can identify the test case that most recently exercised it.

7. Hold your mouse over any green line in the Coverage Viewer.

A tool tip appears showing the name of the test case that exercised this line:

```

begin
  1 1   *      DATABASE_GET_TABLE_RECORD (
    TABLE => TABLE,
    DATA  => TABLE_DATA);

  1 2   *      TABLE_DATA_IS_OCCUPIED := true;
  1 3   *      TABLE_DATA_NUMBER_IN_PARTY := TABLE_DATA.NU
  1 4   *      TABLE_DATA_ORDER(SEAT) := ORDER;
    PLACE_ORDER.001
  1 5   *      ADD_INCLUDED_DESSERT \ TABLE_DATA_ORDER(SEAT);
  1 6   *      case ORDER_ENTREE is
when TYPES.NO_ORDER =>
  1 7   *          null;
when TYPES.STEAK =>
  1 8   *          TABLE_DATA_CHECK_TOTAL := TABLE_DATA.CHI

```



Note: If you had run multiple test cases with coverage enabled, and had multiple check boxes selected in the Environment View, the tool tip would list all cases that provided coverage for the selected line.

8. To access the test case that exercised a line, right-click on the line, then select the test case from the popup menu that appears; in this example, select **PLACE_ORDER.001**:

```

begin
  1 1   *      DATABASE_GET_TABLE_RECORD (
    TABLE => TABLE,
    DATA  => TABLE_DATA);

  1 2   *      TABLE_DATA_IS_OCCUPIED := true;
  1 3   *      TABLE_DATA_NUMBER_IN_PARTY := TABLE_DATA.NU
  1 4   *      TABLE_DATA_ORDER(SEAT) := ORDER;
    Expand Subprograms ▾
    Collapse Subprograms ▾
    PLACE_ORDER.001
  1 5   *      ADD_INCLUDED_DESSERT \ TABLE_DATA_ORDER(SEAT);
  1 6   *      case ORDER_ENTREE is
when TYPES.NO_ORDER =>
  1 7   *          null;
when TYPES.STEAK =>
  1 8   *          TABLE_DATA_CHECK_TOTAL := TABLE_DATA.CHI

```

The test case opens in the Test Case Editor.

9. To return to the Coverage Viewer (the color-coded source code), right-click on **MANAGER** in the **Environment View**, then select **View Coverage** from the popup menu.



Note: You could also select **Window => Coverage Viewers** (Coverage Results for MANAGER).

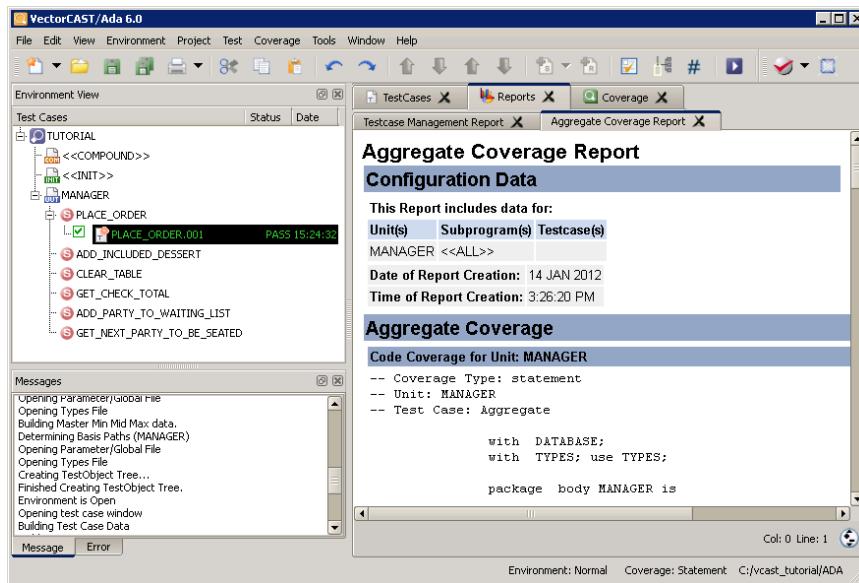
Generating and Managing Coverage Reports

In this section, you will view the Aggregate Coverage Report for your test environment.

An Aggregate Coverage Report includes:

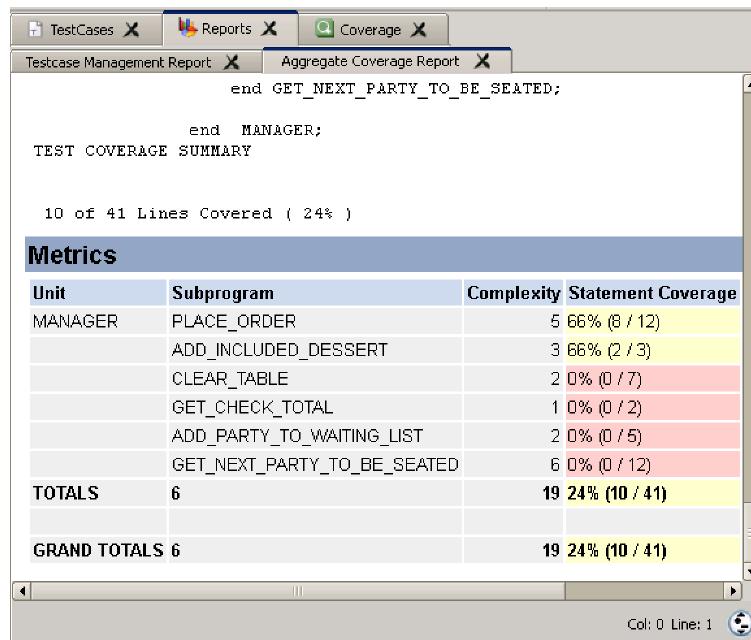
- > A source-listing for each UUT, indicating the lines covered
 - > A metrics table giving the complexity and the level of coverage for each subprogram
1. To view the Aggregate Coverage Report for your test environment, select **Test => View => Aggregate Coverage Report** from the main-menu bar.

The Report Viewer displays a report showing the total coverage for all test cases executed on manager. The associated source code displayed is color-coded to mark the exercised lines (green) and the un-exercised lines (red):



2. Scroll down the report until you come to a table similar to the metrics table you accessed for **PLACE_ORDER** from the Metrics tab.

The partially exercised subprograms, **PLACE_ORDER** and **ADD_INCLUDED_DESSERT**, are shown in yellow; the other subprograms, all unexercised, are shown in red. If 100% coverage for a subprogram is achieved, it shows in green:



3. To print the Aggregate Coverage Report, click the **Print** button on the toolbar , or select **File => Print** from the main-menu bar.
4. To save your report into a file, select **File => Save As**. When the Save As dialog box opens, enter **tutorial_coverage.html** for the file name, then click the **Save** button .
5. You can open your saved report with any HTML browser (Internet Explorer, Mozilla, Netscape, etc.).

5. Choose **File => Close All** in preparation for the next section.

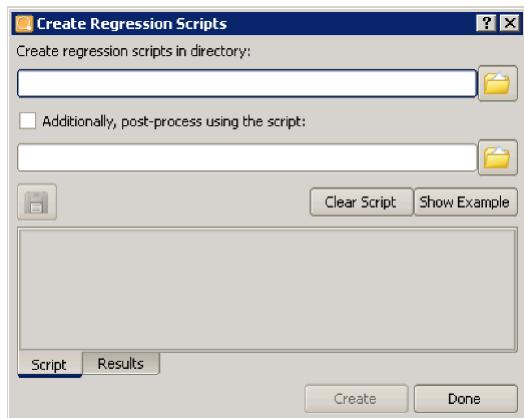
Creating Regression Scripts

Regression Scripts are the files needed to recreate the test environment at a later time. For a typical unit test environment, only 3 files are needed:

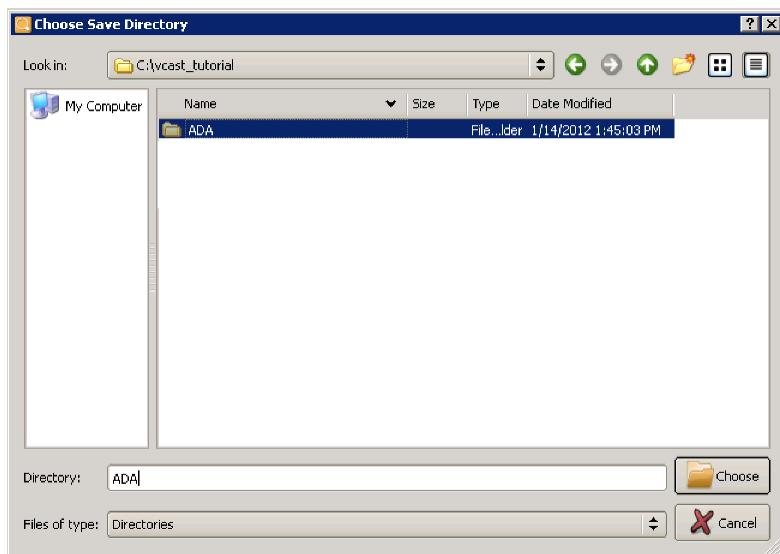
- > A shell script (.bat file on Windows, .sh or .csh on UNIX) sets the options, builds the environment, imports and executes the test cases, and generates reports
- > A test script (.tst) defines test cases to be imported by the shell script.
- > The environment script (.env) defines the settings for the environment, such as the UUT(s), which UUTs are Stub-by-Function, what is the method of stubbing, and where are the search directories.

1. Select **Environment => Create Regression Scripts....**

The Create Regression Scripts dialog appears.

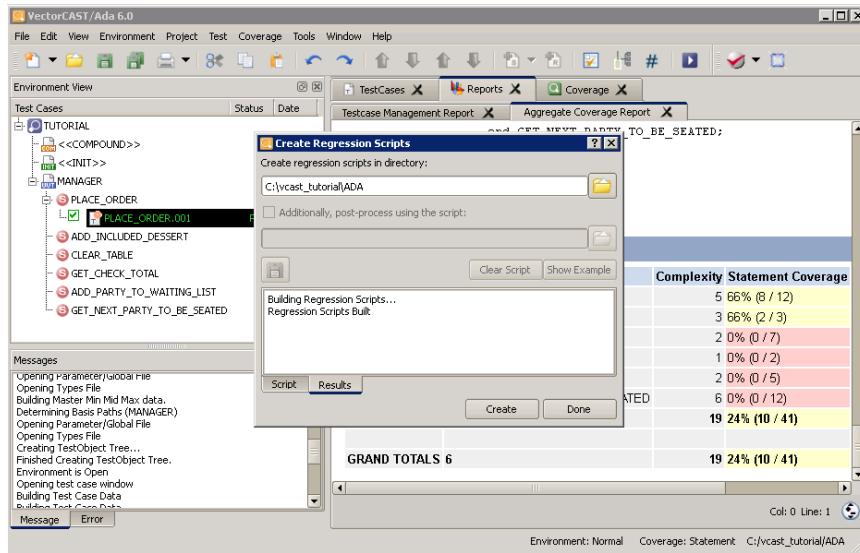


2. In the first edit box, click the **Browse** button and navigate to any directory you choose. For the purposes of this tutorial, we will specify the working directory as the destination for the regression scripts.



3. Click **Create**.

VectorCAST creates the three regression scripts in the location you specified.



- Click **Done** to close the dialog.

Now that you have the regression scripts, you can delete this environment and easily rebuild it at a later time simply by executing the shell script.

Tutorial Summary

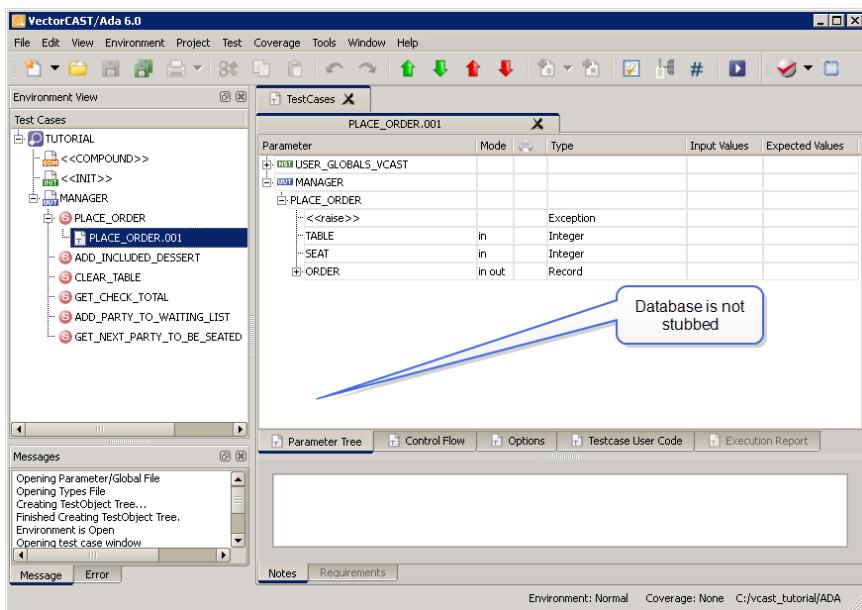
You have now completed the Basic Tutorial. You:

- > Built a test environment named **TUTORIAL** for testing a UUT named **MANAGER**
- > Defined and executed a test case named **PLACE_ORDER.001**
- > Viewed source coverage information
- > Viewed the Test Case Management Report and the Aggregate Coverage Report for your test environment
- > Automatically built additional test cases for exercising all the execution paths through **MANAGER**
- > Achieved 80% statement coverage, and 74% branch coverage
- > Created regression scripts for this environment, which include the shell script, test script, and environment script

Troubleshooting

Problem: Dependents are not stubbed

If you do not see unit **DATABASE** and subprogram **GET_TABLE_RECORD** (as shown below), then you did not stub **DATABASE**.



As a result, the test harness is using the actual unit (DATABASE), which means you cannot specify any input or expected values for subprogram GET_TABLE_RECORD.

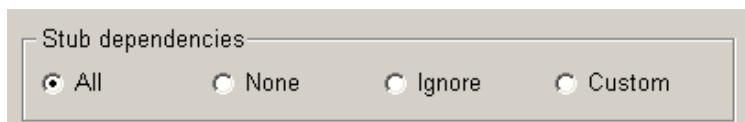
To solve this problem, you can update the environment to change DATABASE from not stubbed to stub:

1. Select **Environment => Update Environment**.

A dialog appears that looks similar to the Create New Environment dialog.

2. Click on **Choose UUTs & Stubs**.

3. Click the radio button "All" under "Stub dependencies" to change DATABASE to be stubbed.



4. Click **Update**.

The environment rebuilds; DATABASE is now stubbed.

Problem: Input values entered instead of expected values

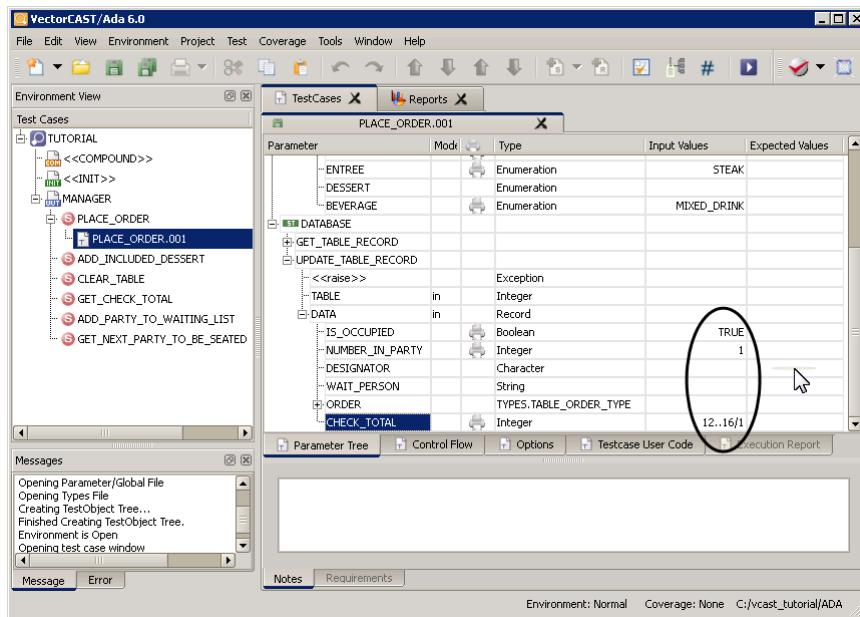
If you enter expected values into Input Test Data fields, you will see the test-case listing as shown below. Note that there is no information under **Expected Test Data**.



File Name: "C-000001.DAT"	
Requirements/Notes	
Input Test Data	
UUT: MANAGER	
Subprogram: PLACE_ORDER	
TABLE (in)	2
SEAT (in)	1
ORDER (in out)	
SOUP	ONION
SALAD	CAESAR
ENTREE	STEAK
DESSERT	
BEVERAGE	MIXED_DRINK
Stubbed Unit: DATABASE	
Subprogram: GET_TABLE_RECORD	
DATA (out)	
NUMBER_IN_PARTY	0
CHECK_TOTAL	0
Subprogram: UPDATE_TABLE_RECORD	
DATA (in)	
IS_OCCUPIED	TRUE
NUMBER_IN_PARTY	1
CHECK_TOTAL	VARY FROM: 12.0000 To:16.0000 BY:1
Expected Test Data	

To solve this problem:

1. Select **File => Close** to close the test-case data report.
2. Double-click PLACE_ORDER.001.
3. Delete the values **true**, **1**, and **12..16** from the Input Values column and enter them into the **Expected Values** column (to the right of the Input column).



You can resize the VectorCAST main window or scroll the Test Case Editor to better view the column, if needed.

Multiple UUT Tutorial

This tutorial demonstrates how to use VectorCAST's multiple UUT feature to conduct integration testing.

When testing more than one UUT in an environment, it is typical that the units interact. VectorCAST's multiple UUT (multi UUT) feature enables you to do this.

In VectorCAST, there are two types of test cases: simple test cases and compound test cases. In the Basic Tutorial, you created and used a simple test case, which corresponded to a single invocation of a UUT. A compound test case is a collection of simple test cases that invokes a UUT in different contexts. The *data is persistent* across these invocations. Data persistence is very important when doing integration testing.

This tutorial will take you through the steps of building a compound test case to test the interaction between the unit **manager** and the unit **database**. The unit **manager** is a simple database in which actual data is stored in data objects defined within the unit. You are guided through the steps of building a compound test case to write data into this database. You then use the same compound test case to retrieve data from the database to verify it.

It is recommended that you review the relevant source listings in Appendix A, "Tutorial Source Code Listings" on page 311 before proceeding.

What You Will Accomplish

In this tutorial you will:

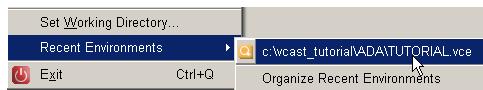
- > Build a multi UUT environment by updating the environment you built in the Basic Tutorial
- > Add a test case to unit **DATABASE**

- > Use the simple test cases to create a compound case
- > Verify the data flow between UUTs **MANAGER** and **DATABASE**
- > Set processing iterations across ranges of input data
- > Build and execute a compound test case

Building a Multi UUT Environment

In this section, you will modify for integration testing the environment you built in the Basic Tutorial.

1. Start VectorCAST if it is not running (refer to "Starting VectorCAST" on page 8).
2. Select **File => Recent Environments**, and then select the environment you created in the first tutorial (named TUTORIAL.vce by default):



Removing PLACE_ORDER.001

The test case present in this environment, PLACE_ORDER.001, was created on the basis of **DATABASE** being a stubbed dependent of **MANAGER**. However, in your new environment, you want **DATABASE** to be a second UUT. Therefore, you must remove PLACE_ORDER.001, because the expected values in the stubbed unit are no longer applicable.

1. To remove PLACE_ORDER.001, right-click on **PLACE_ORDER.001** in the Environment View, and then select **Delete**.

A message box asks you to confirm deletion:



2. Click **Yes**.

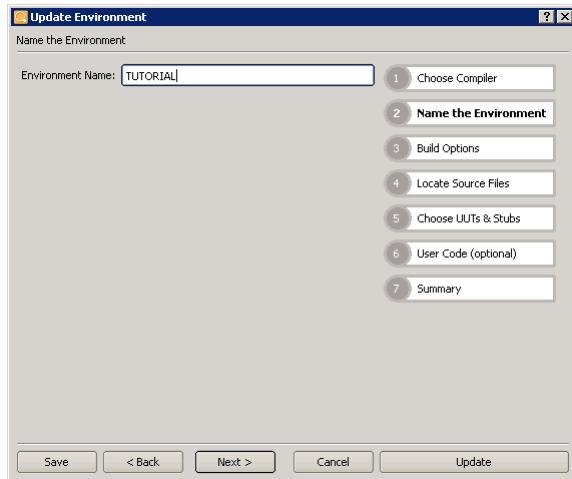
The name PLACE_ORDER.001 disappears from the Environment View.

Updating the Environment

You will now add **DATABASE** to the test environment as a UUT.

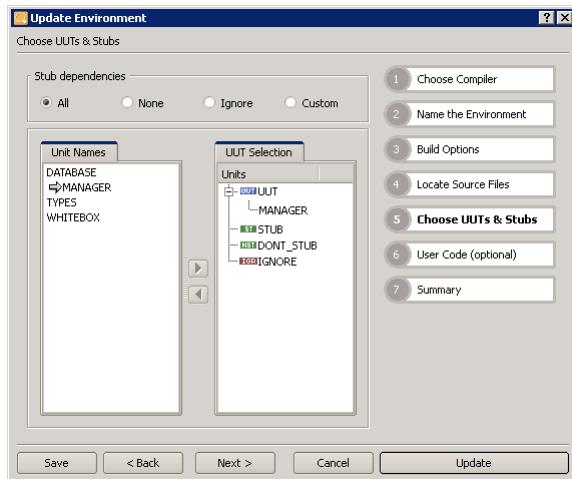
1. Select **Environment => Update Environment**.

The Update Environment wizard appears:



This wizard is similar to the Create Environment wizard you used in the Basic Tutorial to create the original environment. There are no red outlines or accents because there are no errors or missing items of information.

2. Click **Choose UUTs & Stubs**.

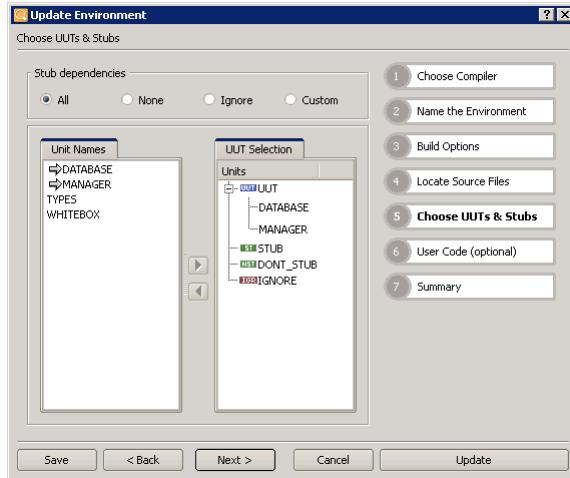


3. In the **Unit Names** pane, double-click **DATABASE**.

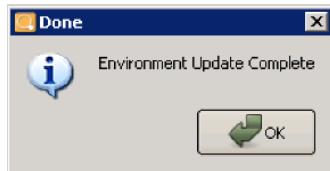


Note: Double-clicking a name in the Unit Names pane is a shortcut for selecting the name and then clicking the right-arrow (▶).

Both **DATABASE** and **MANAGER** are now listed as Units Under Test:

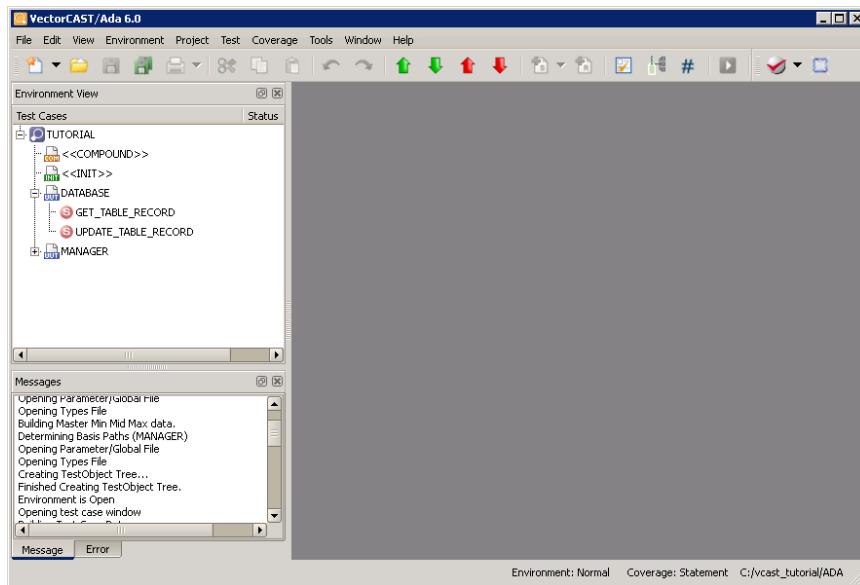


4. Click **Update**.
5. During processing, messages display in the Message window. A message appears confirming that the update has completed:



6. Click **OK**.

The updated environment opens:



Note in the Environment View that **DATABASE** is now listed as a UUT, and also that it has two subprograms: **GET_TABLE_RECORD** and **UPDATE_TABLE_RECORD**.

In the next section, you will create a simple **DATABASE** test case and a simple **MANAGER** test case and then combine these two simple cases to create a compound case.

Building a Compound Test Case

You now have two UUTs to use toward creating a compound test case.

You create a compound test case by creating a separate test case for each subprogram to be included in the test and then aggregating these test cases into a compound test case.

To create the simple test cases, you use the same procedure you used in the Basic Tutorial to create the test case **PLACE_ORDER.001** that you deleted for the purposes of this tutorial.

Insert PLACE_ORDER.001

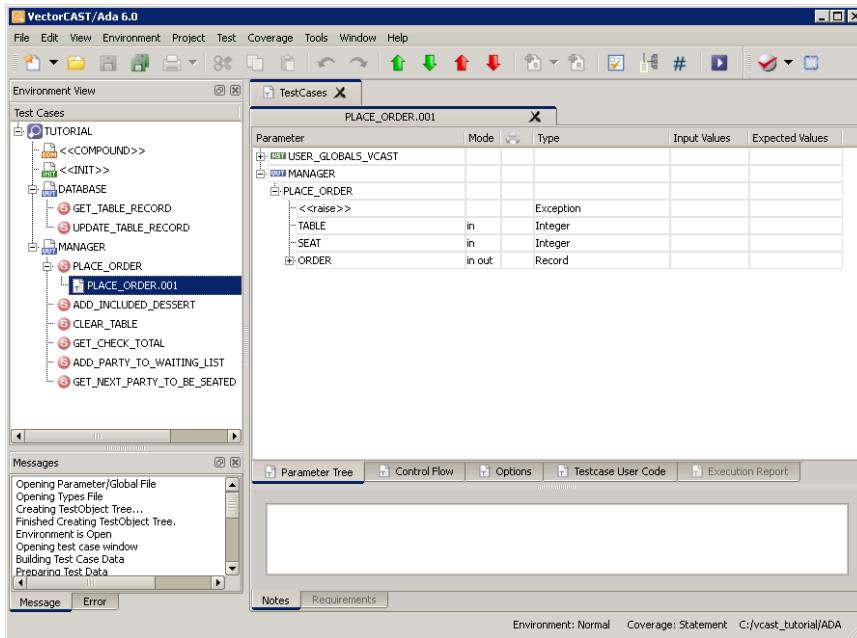
In this section, you will:

- > Create and add Input Values to the test case **PLACE_ORDER.001**. This test case will place an order at a table in the restaurant.
- > Create a simple test case for the subprogram **GET_TABLE_RECORD**, which retrieves data from the database. This test case will be used to verify that **PLACE_ORDER** is updating the database properly.
- > Place these test cases into a compound test case

To create and add order data to **PLACE_ORDER.001**:

1. In the Environment View, expand **MANAGER** into its subprograms, and then expand **PLACE_ORDER** into its test cases.

2. Right-click **PLACE_ORDER** and choose **Insert Test Case**.
 The test case **PLACE_ORDER.001** displays in the Test Case Editor.

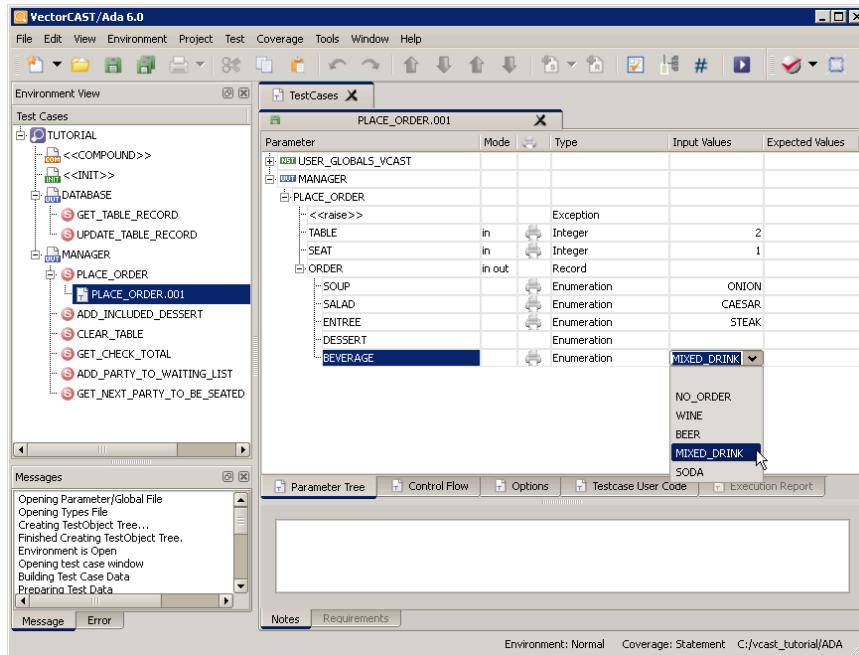


3. Assign Input Values for Table and Seat as you did in the Basic Tutorial:
 Table: **2**
 Seat: **1**
4. Expand **ORDER**.
5. Assign values to the ORDER parameters as indicated below.

The idea is to populate an ORDER record with a set of Input Values you will also use as expected values for a retrieval operation involving **GET_TABLE_RECORD**:

Soup	ONION
Salad	CAESAR
Entree	STEAK
Dessert	(leave unspecified)
Beverage	MIXED_DRINK

These values translate into the following situation: Someone is occupying seat 1 at table 2, and has ordered onion soup, a Caesar salad, a steak entrée, a mixed drink. This combination yields an included dessert of PIE, so we will expect PIE when later verifying the ORDER.



- Save the test case by clicking the **Save** button  on the toolbar.

Because you will be making your test case part of a compound test case, and you only want to use this test within a compound case, you need to designate it as Compound Only. Compound Only test cases can only be executed from within a compound test. They cannot be executed individually or as part of a batch.

In some cases, you will want to use simple test cases both individually and as part of a compound case. In these instances, it is not necessary to apply the Compound Only attribute to the simple cases.

- Right-click **PLACE_ORDER.001** and select **Compound Only**.

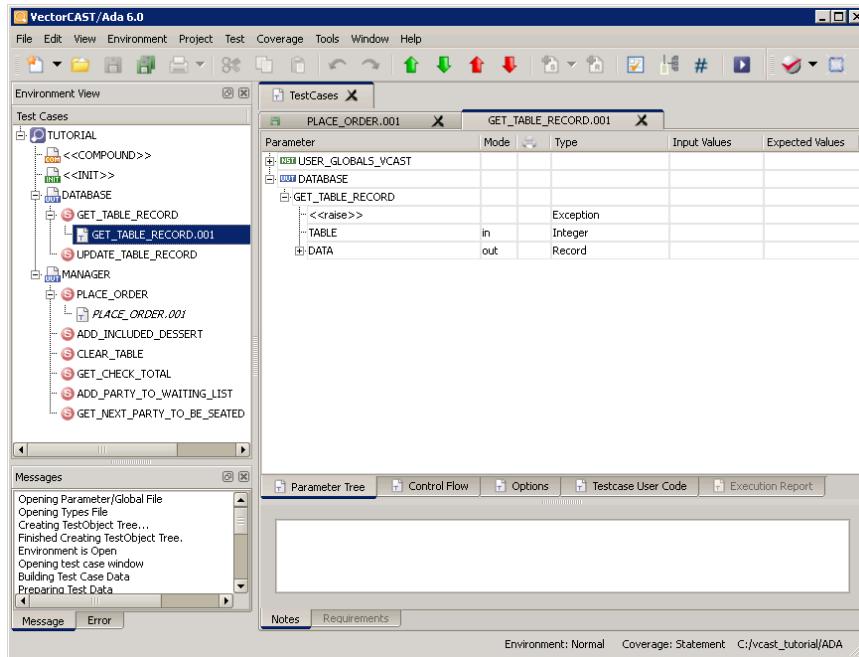
The test case typeface changes to *italic*, signifying that this test case has been designated Compound Only.

Create a Test Case for the GET_TABLE_RECORD Function

In this section, you will create a test case for the function **GET_TABLE_RECORD**. This function retrieves data from the database and verifies it against a set of expected values.

- In the Environment View, expand **DATABASE** into subprograms (if not already expanded).
- Right-click **GET_TABLE_RECORD** and select **Insert Test Case**.

The test case GET_TABLE_RECORD.001 opens in the Test Case Editor:



- Keeping with our example situation, enter **2** as the input value for **TABLE**.



Note: Make sure you enter the value into the Input Values column, not into the Expected Values column. You will enter the expected values for this case in the next step.

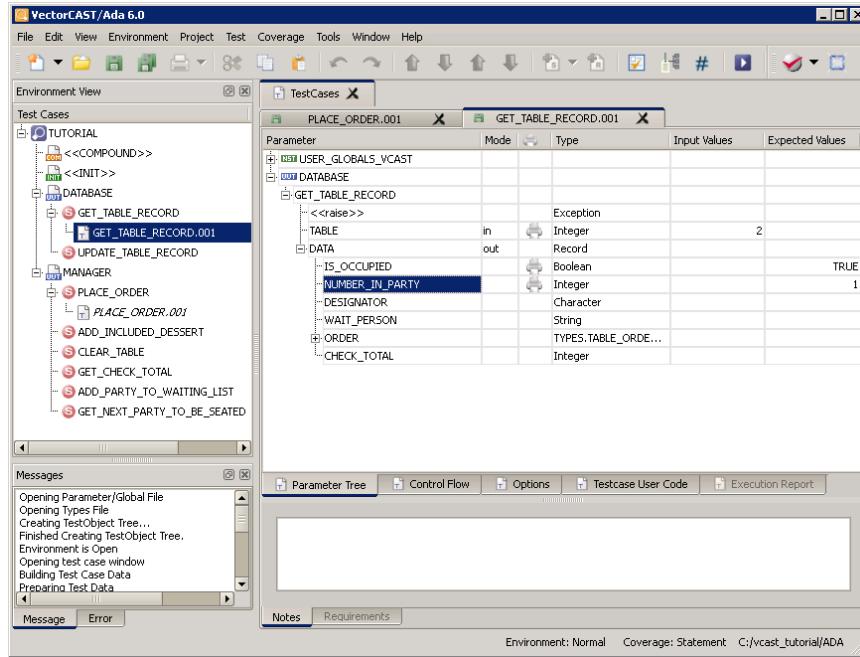
- Expand **DATA** (click plus symbol \oplus), then enter the following Expected Values:

IS_OCCUPIED **TRUE**

NUMBER_IN_PARTY **1**



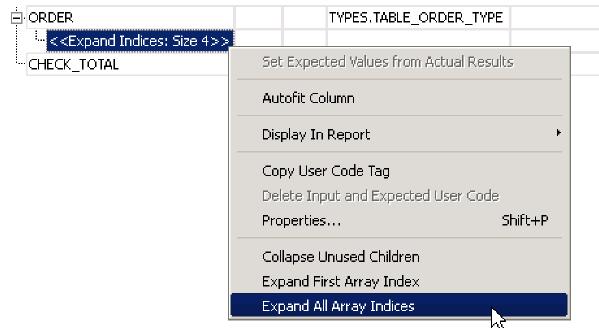
Note: Make sure you enter these values in the **Expected Values** column.



5. Expand **ORDER**.

ORDER is an array of records of size 4.

6. To view the array-indices of **ORDER**, right-click on **<<Expand Indices: Size 4>>** and select **Expand All Array Indices**:



The array expands, displaying the four array elements:

UUT DATABASE					
GET_TABLE_RECORD					
<<raise>>			Exception		
TABLE	in		Integer		2
DATA	out		Record		
IS_OCCUPIED			Boolean		TRUE
NUMBER_IN_PARTY			Integer		1
DESIGNATOR			Character		
WAIT_PERSON			String		
ORDER			TYPES.TABLE_ORDER_TYPE		
<<Expand Indices: Size 4>>					
+ ORDER(1)			Record		
+ ORDER(2)			Record		
+ ORDER(3)			Record		
+ ORDER(4)			Record		
CHECK_TOTAL			Integer		

In this example, you want to verify input values written to the application database. The data consists of an order taken for seat 1 at table 2.

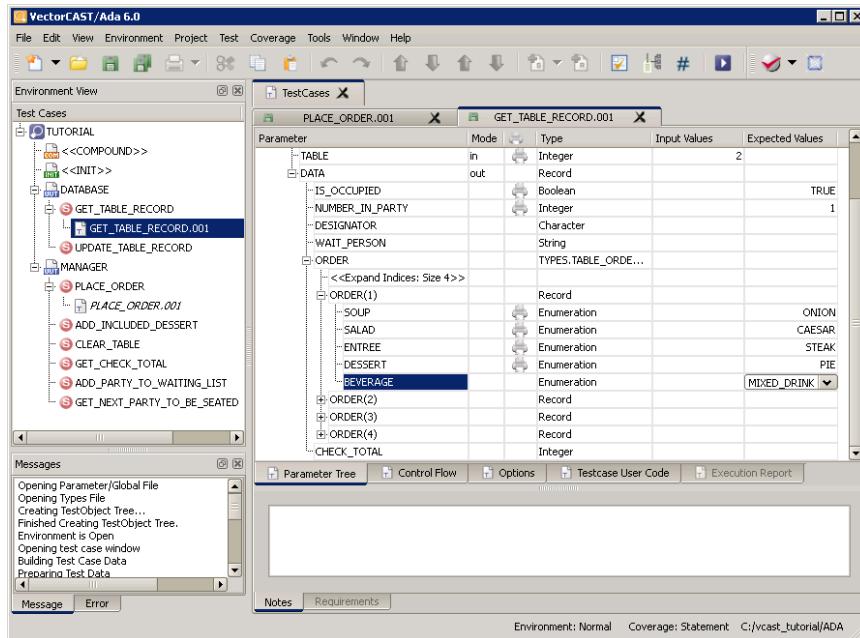
7. Expand **ORDER(1)**.

ORDER(1) corresponds to seat 1. The expected values you specify for ORDER(1) must match the input values you previously specified in PLACE_ORDER.001. If you use any other index, the expected results will not match.

8. Assign the following **Expected Values** to the appropriate fields of **ORDER(1)**:

SOUP	ONION
SALAD	CAESAR
ENTREE	STEAK
DESSERT	PIE
BEVERAGE	MIXED_DRINK

 **Note:** Make sure you enter these values into the **Expected Values** column.



9. Click **Save** button .

Note that ORDER(02), ORDER(03), and ORDER(04) are not populated with data. Populating a single ORDER record is sufficient to verify the operation of the two subprograms under test.

10. Right-click **GET_TABLE_RECORD.001** in the Environment View, then select **Compound Only**.

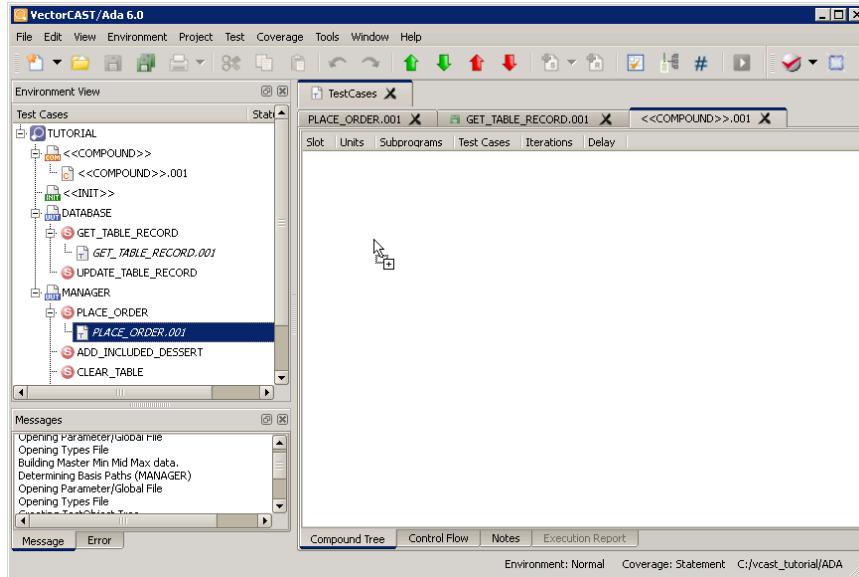
You now have two test cases designated Compound Only. The first case, **PLACE_ORDER.001** places a dinner order at table 2, seat 1. This order is stored in the application database by way of **PLACE_ORDER** calling **UPDATE_TABLE_RECORD**. The second case, **GET_TABLE_RECORD.001** retrieves the order from the database for verification.

You are now ready to use these two simple test cases to create a single compound test case.

Creating a Compound Test Case

If you were to execute your test cases individually, they would fail, because the test harness would be called separately for each test case, and the data would not be retained in the data structure between the two calls. In order for your test cases to work together, they must be combined into a compound case.

- In the Environment View, right-click **<<COMPOUND>>** and select **Insert Test Case**. A test case named **<<COMPOUND>>.001** appears under **<<COMPOUND>>**, and is selected (highlighted) by default.
- Click and hold the cursor on **PLACE_ORDER.001** in the Environment View; drag until the plus symbol  appears in the Test Case Editor; release.



An entry appears in the Test Case Editor (slot 1) as a member of test case <<COMPOUND>>.001.

3. Do the same with test case **GET_TABLE_RECORD.001**.

Your compound test case now has two slots: PLACE_ORDER.001, and GET_TABLE_RECORD.001:

Slot	Units	Subprograms	Test Cases	Iterations	Delay
1	MANAGER	PLACE_ORDER	PLACE_ORDER.001	1	0
2	DATABASE	GET_TABLE_RECORD	GET_TABLE_RECORD.001	1	0

4. Click **Save**.

You are now ready to execute your compound test case.

5. In the Environment View, select <<COMPOUND>>.001, then click **Execute** button  on the toolbar to execute it.

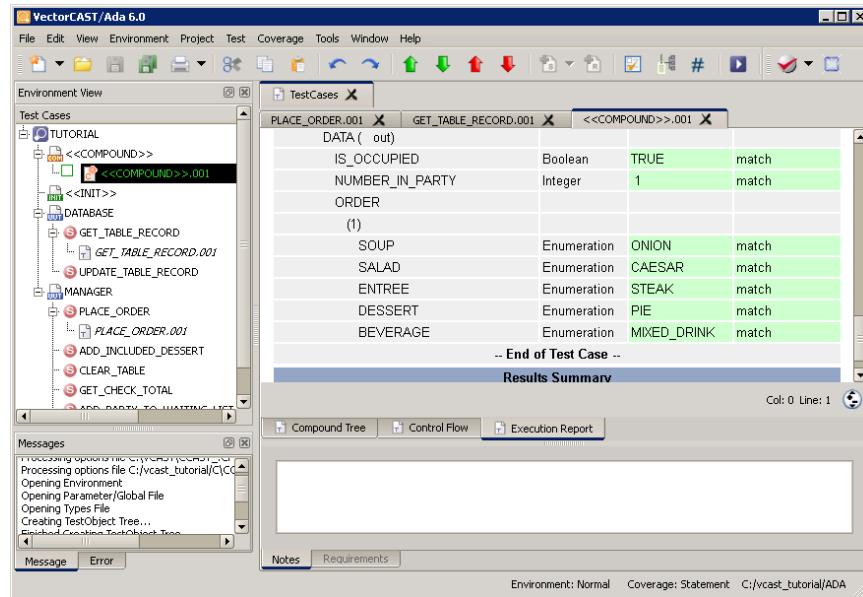
An execution report for the compound test case is displayed in the Test Case Editor.



Note: If your compound test case fails, make sure PLACE_ORDER.001 uses parameters table 2, seat 1, and that GET_TABLE_RECORD.001 uses table 2 and the proper expected values in ORDER(1).

6. Scroll down the report.

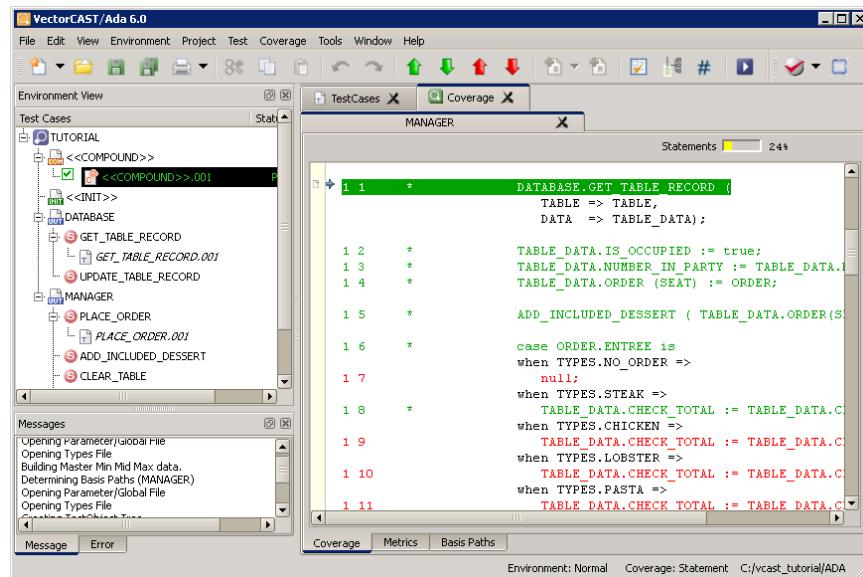
Note that the expected values you specified were matched against corresponding actual values, verifying the data flow from **MANAGER** to **DATABASE**.



- Click the checkbox preceding <<COMPOUND>>.001 in the Environment View:



The tab for **MANAGER** opens in the Coverage Viewer.



In the Coverage Viewer, note that the statement assigning the value PIE to ORDER.DESSERT

is green, indicating that it was covered by test execution.

Initializing Branch Coverage

To find out which branches are covered by executing this compound test case:

1. Choose **Coverage => Initialize => Branch**.

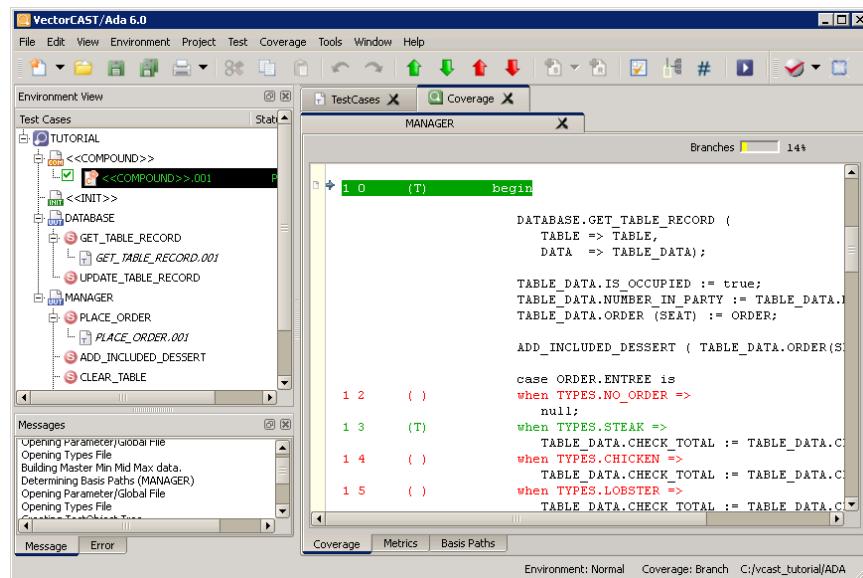
The status bar shows “Coverage: Branch” and the checkbox next to <<COMPOUND>>.001 is removed, because the statement results were made invalid during coverage initialization. To see branch coverage achieved, you must execute the test case again.

Note that the *execution results* were not made invalid during coverage initialization.

2. Select <<COMPOUND>>.001 in the Environment View and click the **Execute** button  in the toolbar.
3. Click the checkbox preceding <<COMPOUND>>.001 in the Environment View:



In the Coverage Viewer, the instances of the '(T)' symbol indicate that two PLACE_ORDER branches were covered: the first at the subprogram entry point; the second at the STEAK branch in the case statement.



VectorCAST allows you to build more complex tests by linking multiple simple test cases together. This feature allows you to test threads of execution while maintaining the data common to the various units and subprograms.

Adding Test Iterations for Fuller Coverage

You can reach fuller coverage of **MANAGER** by having each seat at table 2 order a different entrée while

keeping the other order items constant. Varying the Entree value ensures that each branch in the case statement (except for the default case) will be exercised by the test.

In this section, you will modify test case PLACE_ORDER.001 to iterate over the 4 seats at table 2 while iterating over the four entrées. The net effect will be to assign a different entrée to each seat.

1. In the Environment View, double-click **PLACE_ORDER.001**.

The test case PLACE_ORDER.001 opens in the Test Case Editor.

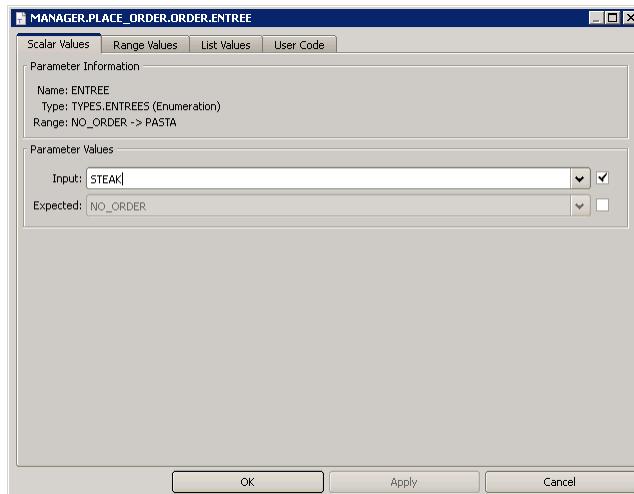
2. Enter **1..4** as the input value for **Seat**; press **Enter**.

Your entry changes to **1..4/1**. The **/1** indicates that the input value for Seat will iterate over its range with a delta of 1. (It is possible to specify a delta other than 1.)

UUT MANAGER		
PLACE_ORDER		
<<raise>>		Exception
TABLE	in	Integer 2
SEAT	in	Integer 1..4/1
ORDER	in out	Record
SOUP		Enumeration ONION
SALAD		Enumeration CAESAR
ENTREE		Enumeration STEAK
DESSERT		Enumeration
BEVERAGE		Enumeration MIXED_DRINK

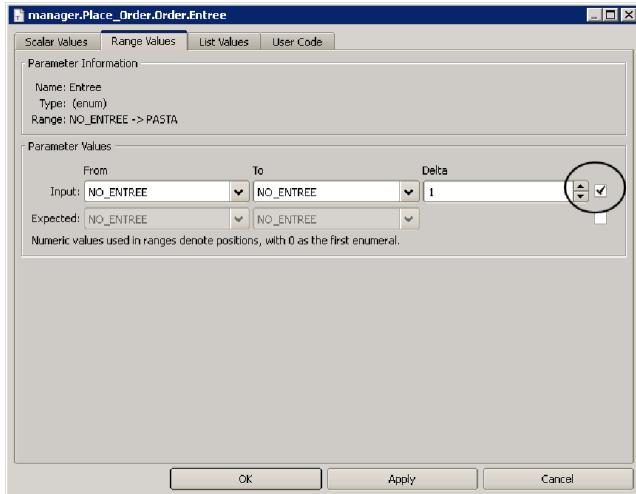
3. Double-click **ENTREE**.

A dialog appears for setting parameter specifications:

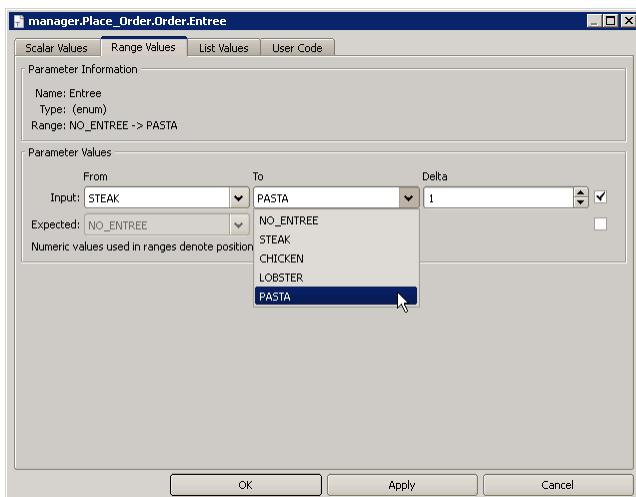


4. Click the **Range Values** tab.

5. Enable the **Input** fields by clicking the checkbox at the far right:



6. Select STEAK for From; PASTA for To; leave Delta at 1:



7. Click OK.

8. Click Save button on the Toolbar 

Parameter	Type	Input Values	Expected Values
<<GLOBAL>>			
UUT manager			
<<GLOBAL>>			
Place_Order			
-- Table	unsigned short	2	
-- Seat	unsigned short	0..3/1	
-- Order	struct		
-- Soup	enum	ONION	
-- Salad	enum	CAESAR	
-- Entree	enum	STEAK..PASTA/1	
-- Dessert	enum		
-- Beverage	enum	MIXED_DRINK	

You have now assigned input ranges to two parameters. By default, when two parameters are assigned input ranges, they will iterate in *parallel*, which is what you want to occur in this case. These two parameters will iterate with the following value pairs:

- Seat = 1 Entree = Steak
- Seat = 2 Entree = Chicken
- Seat = 3 Entree = Lobster
- Seat = 4 Entree = Pasta

Given this information, you can specify expected values in your compound test case for each seat.

9. Click the tab for test case **GET_TABLE_RECORD.001** at the top of the Test Case Editor. The parameter tree for GET_TABLE_RECORD.001 comes to the front. You need to edit this tree to reflect the expected results for your compound test. You now need to specify expected values for each of the other three seats at table 2.
10. Expand **Order(2)**. (If you have recently re-opened the TUTORIAL environment, then the unused array elements have been collapsed. If Order(2) is not visible, simply right-click on <<Expand Indices: Size 4>> and choose **Expand All Array Indices**.) Order(2) maps to seat 1. Specify the following expected values for seat 2:

- | | |
|----------|------------------------------|
| Soup | ONION |
| Salad | CAESAR |
| Entree | CHICKEN |
| Dessert | NO_DESSERT (see Note) |
| Beverage | MIXED_DRINK |

 **Note:** For Order(2), Order(3), and Order(4), Place_Order calls ADD_INCLUDED_DESSERT with a soup, salad, and entrée combination that does not qualify for PIE, so we expect NO_ORDER for DESSERT.

11. Expand **Order(3)**. Enter the following expected values for seat 3:
- | | |
|------|--------------|
| Soup | ONION |
|------|--------------|

Salad	CAESAR
Entree	LOBSTER
Dessert	NO_ORDER
Beverage	MIXED_DRINK

12. Expand **Order(4)**.

Enter the following expected values for seat 4:

Soup	ONION
Salad	CAESAR
Entree	PASTA
Dessert	NO_ORDER
Beverage	MIXED_DRINK



You now have ORDER data for each seat at table 2. Note that these orders are identical in each case except for the Entree and the Dessert. Only the ENTREE variable is needed to exercise each branch in the **case** statement.

ORDER	TYPES.TABLE_ORDER_TYPE
<<Expand Indices: Size 4>>	
ORDER(1)	Record
SOUP	Enumeration
SALAD	Enumeration
ENTREE	Enumeration
DESSERT	Enumeration
BEVERAGE	Enumeration
ORDER(2)	Record
SOUP	Enumeration
SALAD	Enumeration
ENTREE	Enumeration
DESSERT	Enumeration
BEVERAGE	Enumeration
ORDER(3)	Record
SOUP	Enumeration
SALAD	Enumeration
ENTREE	Enumeration
DESSERT	Enumeration
BEVERAGE	Enumeration
ORDER(4)	Record
SOUP	Enumeration
SALAD	Enumeration
ENTREE	Enumeration
DESSERT	Enumeration
BEVERAGE	Enumeration

You must now change the number of occupied seats at table 2 to reflect the current test case.

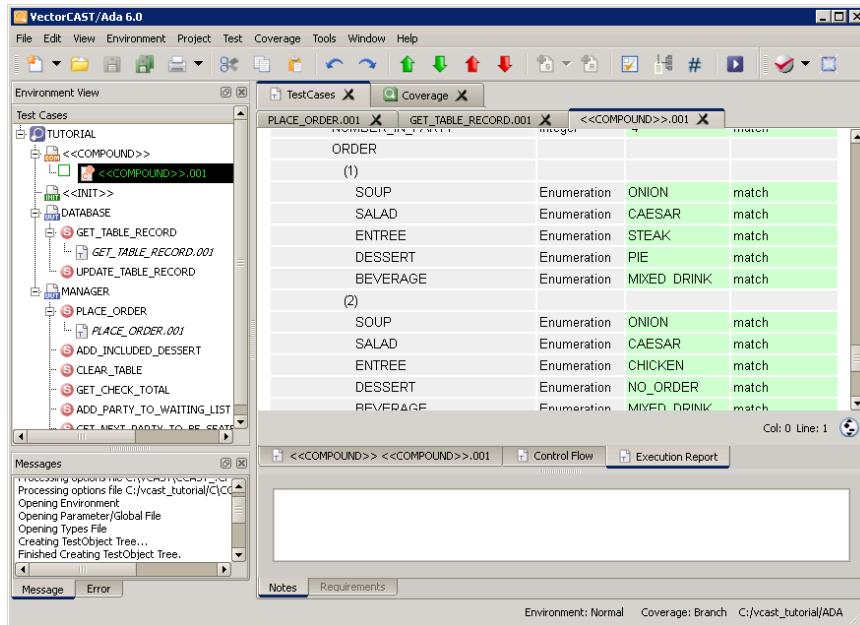
13. Scroll up to **DATA** and change **NUMBER_IN_PARTY** to **4**.

14. Save.

15. Select **<<COMPOUND>>.001** in the Environment View; then click

<<COMPOUND>>.001 turns green; your test has passed. An execution report for your compound test displays in the Test Case Editor.

16. Scroll down the report to view the green-coded results for each **ORDER** element:



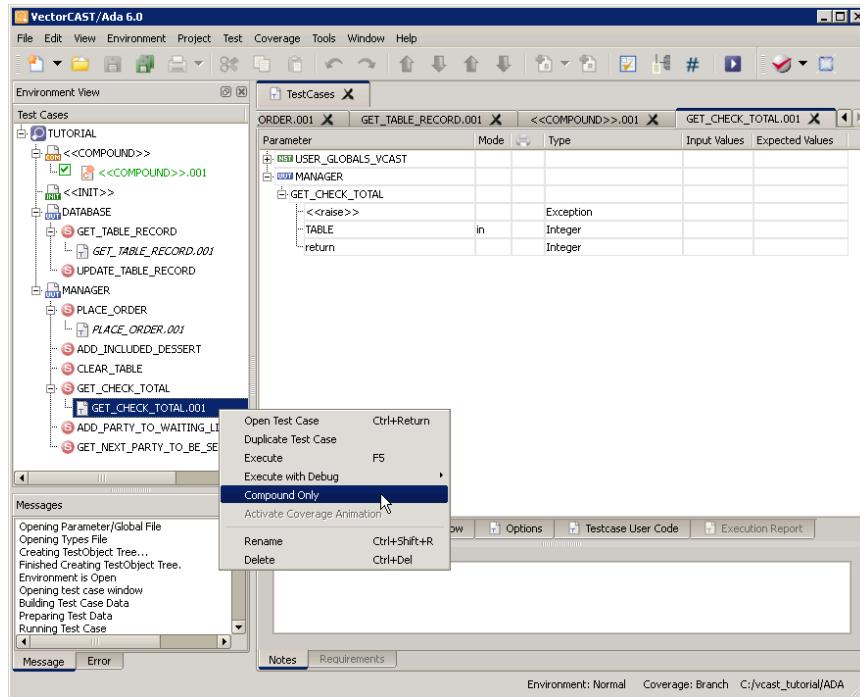
Adding Test Cases to a Compound Test Case

In this section, you will verify the check total for the four entrée orders at table 2 by adding a third test case to your compound case.



Note: The amount of instruction given in this section will be a little less than in the previous sections.

- Under the MANAGER unit, insert a test case for **GET_CHECK_TOTAL** and make it **Compound Only**.

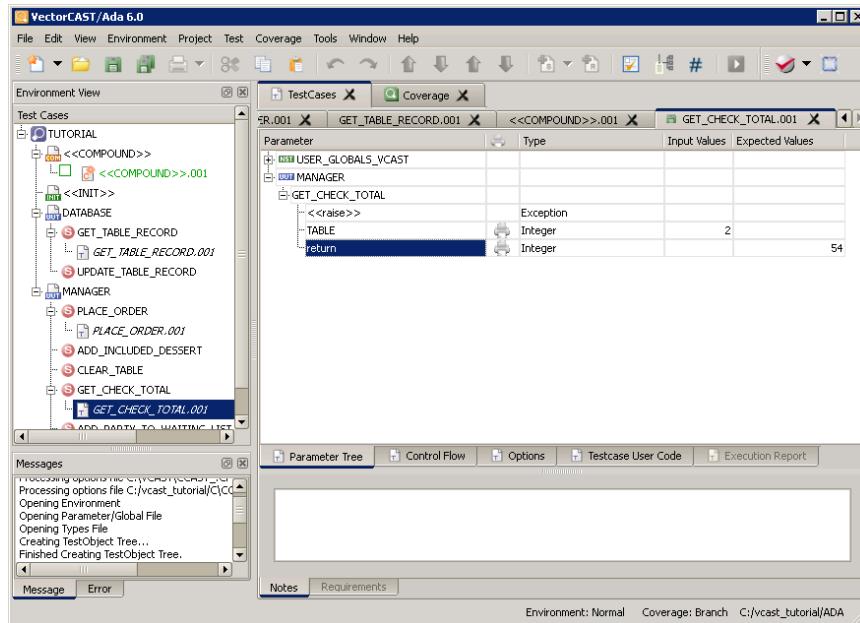


2. Enter **2** as the input value (not expected value) for **TABLE**.

On the basis of the entrée prices coded into the application, you expect the check to total \$54 (for one of each entrée).

3. Enter **54** as the expected value (not input value) for **return**.

4. Save.

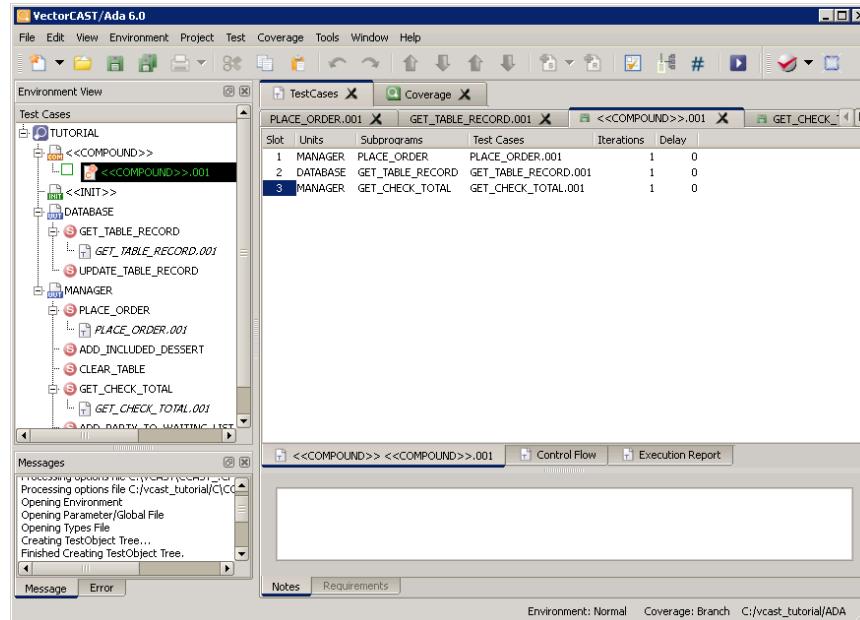


You can now add this test case to the compound case.

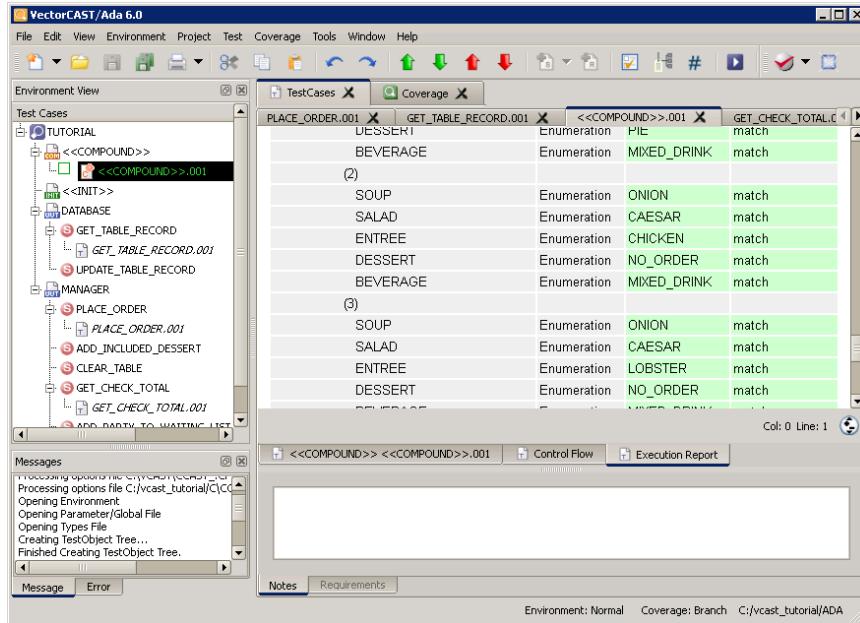
Click the <<COMPOUND>>.001 tab at the top of the Test Case Editor to bring <<COMPOUND>>.001 to the top.

5. Click the **Compound Tree** tab along the bottom of the Test Case Editor. The Test Case Editor displays the slots in test case <<COMPOUND>>.001.
6. To add test case GET_CHECK_TOTAL.001, click and hold the cursor on **GET_CHECK_TOTAL.001** in the Environment View; drag until the symbol appears in the Test Case Editor; release.

GET_CHECK_TOTAL.001 displays as slot 3 in your compound test case:

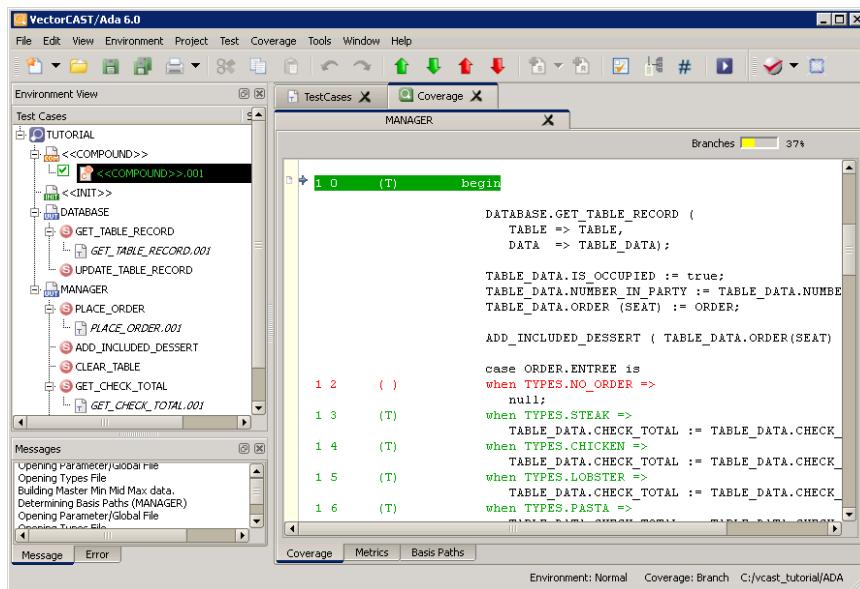


7. Save.
8. Select <<COMPOUND>>.001 in the Environment View, then click to execute it. <<COMPOUND>>.001 turns green in the Environment View; an execution report displays in the Test Case Editor.
9. Scroll down the report to view the green-shaded match results.



- To view the coverage data, click the green **checkbox** preceding <<COMPOUND>>.001 in the Environment View.

The Coverage Viewer opens:



- Scroll down.

```

case ORDER.ENTREE is
  1_2      ( )
    when TYPES.NO_ORDER =>
      null;
  1_3      (T)
    when TYPES.STEAK =>
      TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_
  1_4      (T)
    when TYPES.CHICKEN =>
      TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_
  1_5      (T)
    when TYPES.LOBSTER =>
      TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_
  1_6      (T)
    when TYPES.PASTA =>
      TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_
end case;

DATABASE.UPDATE_TABLE_RECORD (
  TABLE => TABLE,
  DATA  => TABLE_DATA);

end PLACE_ORDER;

procedure ADD_INCLUDED_DESSERT ( ORDER : in out TYPES.ENTREE )
begin
  if ORDER.ENTREE   = STEAK and
    ...

```

Notice that by iterating over four entrées, you exercised four of the five case branches. The status bar at the top right of the Coverage Viewer tells you that total branch coverage was 10 of 27, or 37%.

Exporting a Test Script

1. Click on the top level of the environment, **TUTORIAL**, in the Environment View.
2. Select **Test => Scripting => Export Script...**, and give the test script the name **TUTORIAL_MULTI.tst**.
3. Click **Save** to create the test script.

Tutorial Summary

This tutorial built on the environment you created in the Basic Tutorial. You changed one of the dependent units from a smart stub into a UUT. You then added test cases to both units, and combined these test cases into a Compound test case, which you used to test the data flow from **MANAGER** to **DATABASE**. When test cases are combined into a Compound test case, the data remains persistent between them.

In sum, you:

- > Built a multi UUT environment by updating the environment you built in the Basic Tutorial
- > Added a test case for the subprogram **PLACE_ORDER** in **MANAGER**
- > Added a test case for the subprogram **GET_TABLE_RECORD** in **DATABASE**
- > Used these simple test cases to create a compound case
- > Verified the data flow between UUTs **MANAGER** and **DATABASE**
- > Set up range iterations for a scalar parameter and an enumeral
- > Added a test case to an existing compound case

Whitebox Tutorial

This tutorial demonstrates how to use the VectorCAST Whitebox utility. Using Whitebox circumvents the normal data and subprogram visibility issues associated with testing Ada programs by providing direct visibility to all hidden subprograms and data objects. Without this utility, you would have to drive all processing of hidden subprograms from those subprograms that are visible to external packages.

This tutorial walks you through the steps necessary to perform a whitebox conversion of a unit called WHITEBOX. WHITEBOX contains a visible subprogram and a private type declaration in the package specification. It also contains two subprograms and several objects declared in the package body.



Note: This tutorial assumes you have run at least one of the previous Ada tutorials, and that you have set `vcast_tutorial\ADA` (Windows format) as the working directory for the Ada tutorials.

The whitebox utility preprocesses a UUT so that VectorCAST can directly access the subprograms and objects that exist in the body of the UUT. It also allows you to manipulate parameters and objects based on private types defined in the UUT.

The source files for the WHITEBOX package are provided in "Ada" on page 326 . You may wish to review these source files before running the tutorial.

The following code fragment from Appendix A shows the procedure `init_day`. Notice that this procedure is hidden by being declared in the body of the package WHITEBOX. Normally this procedure would not be directly callable from any other external package:

```
package body WHITEBOX is

    --Type Declarations
    type color is (red, green, blue);
    type day is (monday, tuesday, wednesday, thursday);
    --Hidden Procedure Specifications
    procedure init_day (val : day);

    --Hidden Procedure Specifications
    procedure init_color (val : color);

    --Local Object Declarations
    current_day : day;
    current_color : color;

    procedure initialize (pointer : in out pointer_type) is
    begin
        init_day (day'first);
        init_color (color'first);
        pointer.data_index := 1;
        pointer.data_value := 12;
    end initialize;

    procedure init_day (val : day) is
```

```

begin
  current_day := val;
end init_day;

procedure init_color (val : color) is
begin
  current_color := val;
end init_color;

end WHITEBOX;

```

What You Will Accomplish

In this tutorial, you will perform a whitebox conversion on a unit named WHITEBOX. WHITEBOX contains a visible subprogram and a private type declaration in the package specification. It also contains two subprograms and several objects declared in the package body.

Building a Whitebox Test Environment

In this section, you will create a test environment for a UUT named WHITEBOX.

You will:

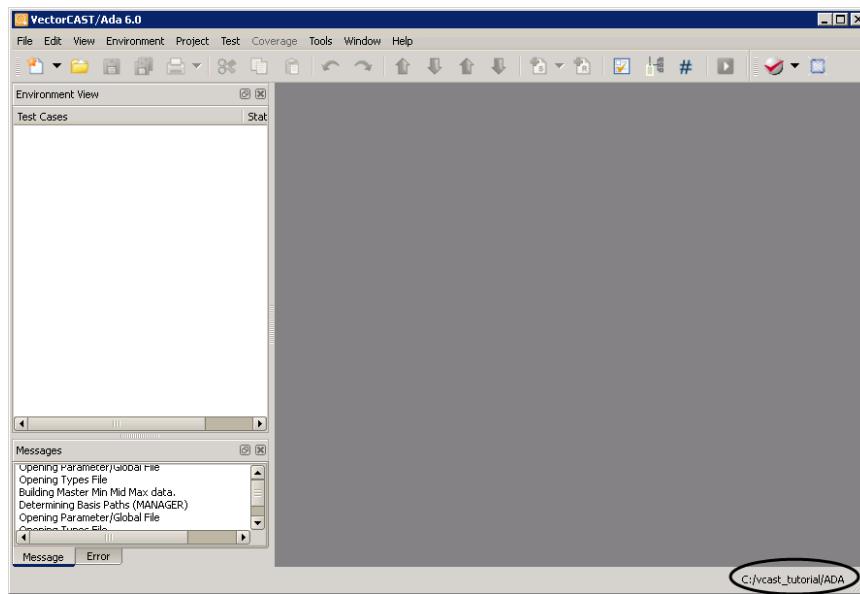
- > Specify a working directory
- > Specify a compiler environment (Ada Host)
- > Specify the source files to be included in the environment
- > Designate the UUT and any stubs
- > Turn on whitebox
- > Build the test harness

Specifying a Working Directory

The working directory contains the files you will use to build your environment. For the purposes of this tutorial, the working directory is **/vcast_tutorial/ADA**.

1. Start VectorCAST if it is not already running, or close the environment from the last tutorial by selecting **File => Close Environment**.
2. If necessary, set the working directory to **/vcast_tutorial/ADA** by selecting **File => Set Working Directory**.

The location and name of the current working directory appear at the lower right-hand corner of the VectorCAST main window:

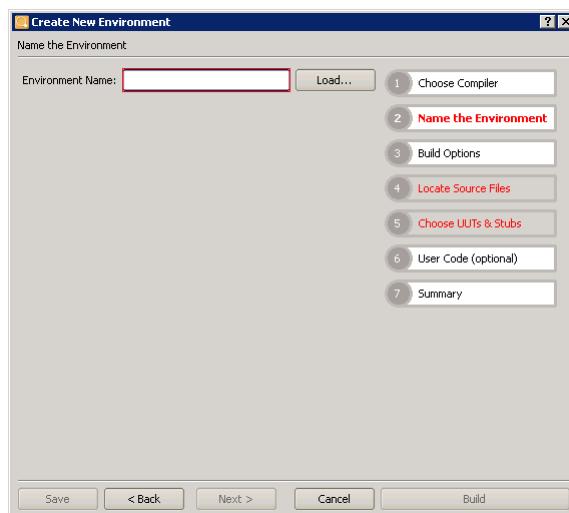


Specifying the Type of Environment to Build

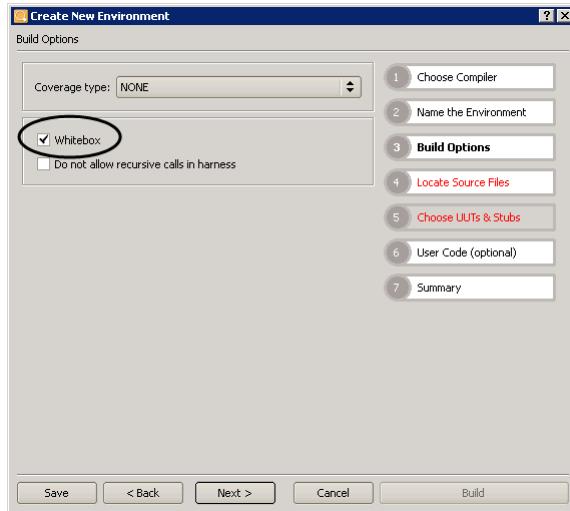
1. To begin creating an environment, click the **New** button  on the toolbar, then select **Ada Host Environment** from the drop-down menu.

The Create New Environment wizard opens.

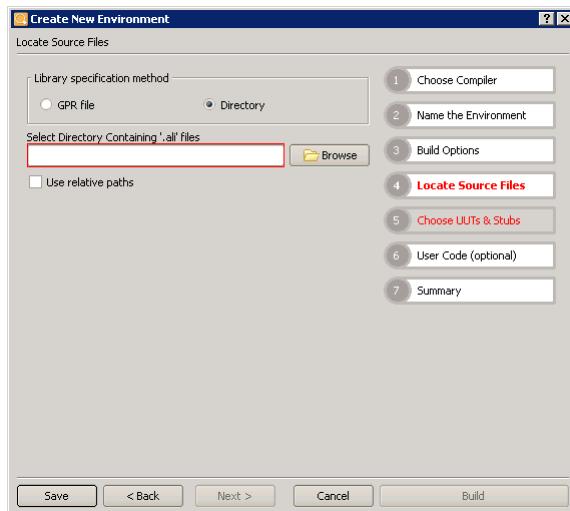
If you have previously run one of the other Ada tutorials, the wizard opens to the Name the Environment page. If you have not previously run one of the other Ada tutorials, the wizard opens to the Choose Compiler page. (If you need to select or change the compiler, select the **Choose Compiler** page now.)



2. On the Name the Environment page, enter **white** into the Environment Name field. VectorCAST echoes your entry in capital letters.
3. Click **Next**.
4. Check the box preceding **Whitebox**.



5. Click **Next**.
The Locate Source Files page opens:



Specifying the source files to be tested

You use the Locate Source Files page to specify the source files to be tested. In this example, you will be testing a file named WHITEBOX. You previously copied this file into the working directory.

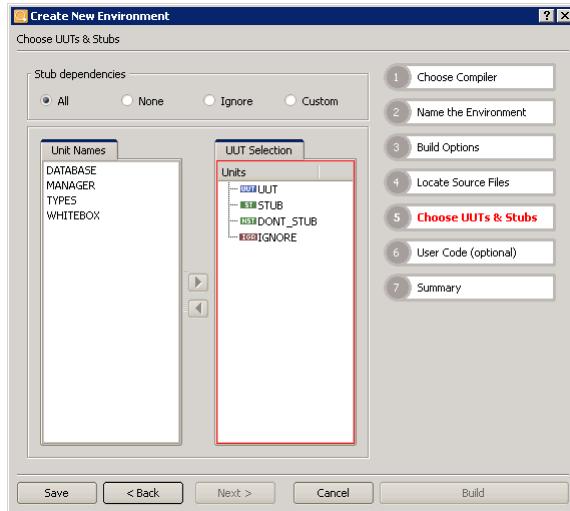
1. Click the **Browse** button .

The Choose Directory dialog appears.

Because you are building the environment in the same location into which you copied the tutorial files, the directory shown in the Look In field (`C:\vcast_tutorial\ADA`) is the correct location.

2. Click **Choose**, then **Next**.

The Choose UUTs & Stubs page appears:



Designating UUTs and stubs

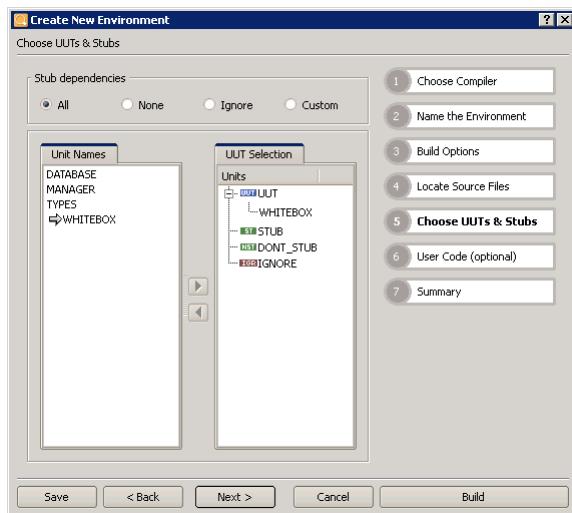
The source files (units) found in the Parent Library by VectorCAST are listed in the Unit Names pane. The unit you want to test in this tutorial is WHITEBOX.

1. In the Unit Names pane, double-click **WHITEBOX**.



Note: Double-clicking a name in the Unit Names pane is a shortcut for selecting the name and then clicking the right-arrow (▶). You can also drag-and-drop.

WHITEBOX moves to the UUT Selection pane.



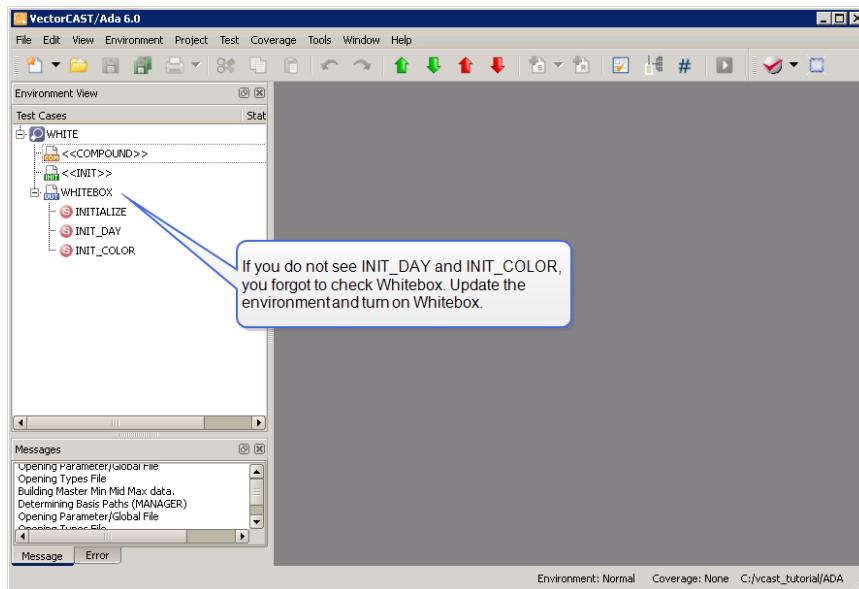
Compiling your specifications into an executable test harness

For the purposes of this tutorial, you will not add any user code to your environment.

You will now build your specifications into an executable test harness.

1. Click the **Build** button.

When processing completes, the window for your new environment opens, as shown below:



The top item listed in the Environment View is the name of the current environment (WHITE). The only UUT is WHITEBOX, which is shown to have three subprograms: INITIALIZE, INIT_DAY, and INIT_COLOR. Hidden procedures INIT_DAY and INIT_COLOR are now visible.



Note: You now have direct visibility to all subprograms, whether they were declared in the specification or in the body of package WHITEBOX.

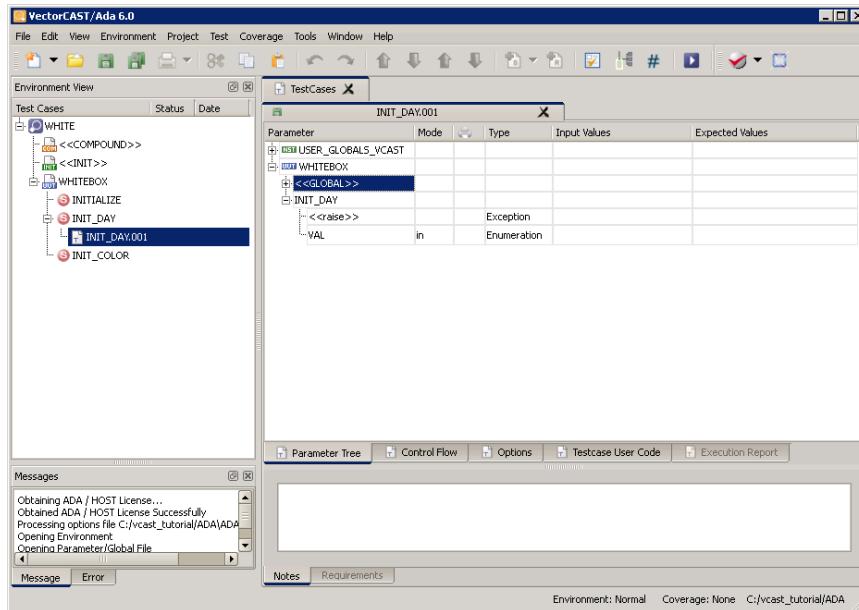
Creating a Test Case for a Hidden Subprogram

In this section, you will create a test case for hidden subprogram INIT_DAY.



Note: Parameter VAL of type DAY, which is defined in the body of package WHITEBOX, is now visible, and can be used as input to INIT_DAY.

1. Insert a test case for the subprogram **INIT_DAY**:



2. In the parameter tree, select **MONDAY** as the input value for the parameter **VAL**.

+ USER_GLOBALS_VCAST				
- WHITEBOX				
+ <<GLOBAL>>				
- INIT_DAY				
<<raise>>		Exception		
VAL	in	Enumeration	MONDAY	

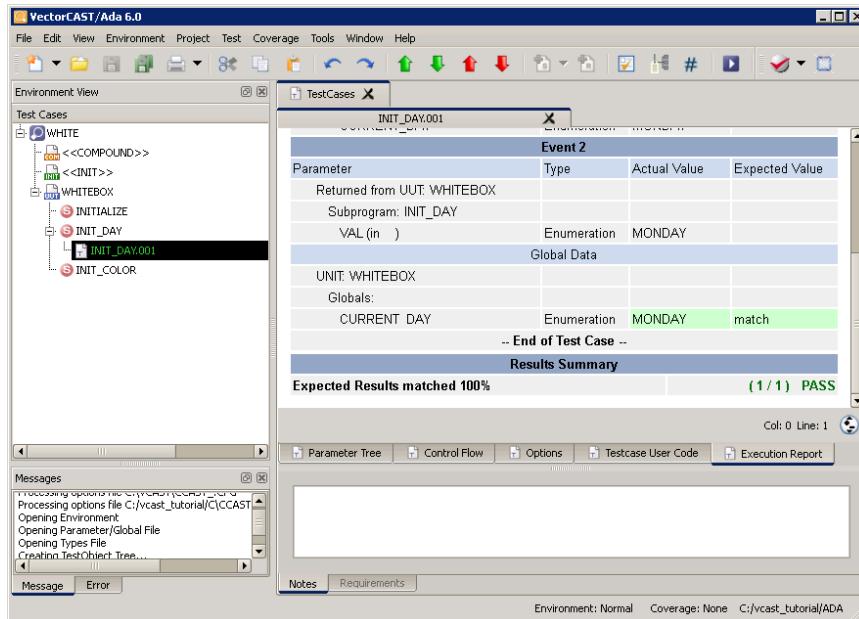
You can now initialize a global object hidden in the body of package WHITEBOX to see how it is affected.

3. Expand **<<GLOBAL>>** and set **MONDAY** as the expected value for **CURRENT_DAY**. Global object CURRENT_DAY of type DAY, which is defined in the body of package WHITEBOX, is now visible.

+ USER_GLOBALS_VCAST				
- WHITEBOX				
+ <<GLOBAL>>				
- CURRENT_DAY		Enumeration		MONDAY
CURRENT_COLOR		Enumeration		
- INIT_DAY				
<<raise>>		Exception		
VAL	in	Enumeration	MONDAY	

4. Save the test case.
5. Click the **Execute** button on the toolbar.

An execution report appears in the Test Case Editor:



Tutorial Summary

By gaining access to all subprograms directly with the Whitebox utility, you can circumvent the visibility testing difficulties inherent to Ada.

Cover Tutorials

Introduction

The tutorials in this chapter demonstrate how to use VectorCAST/Cover to analyze code coverage. You should run these tutorials in the order presented.

Using VectorCAST/Cover to analyze code coverage involves:

1. Building a coverage environment which instruments your source code
2. Building your application to be tested and analyzed using the instrumented source code
3. Generating coverage results
4. Viewing these results

For details on all four steps, refer to "VectorCAST/Cover Overview" on page 15.

C++ Cover Tutorial

This tutorial steps you through the full process of using VectorCAST/Cover to analyze code coverage. You will create a VectorCAST/Cover environment, instrument source files, and compile a sample application. You'll use that sample application to generate test results, which you will then add to the cover environment.

The source modules you will use in this tutorial are components of a simple order-management application for restaurants. The listings for this application are available in Appendix A, "Tutorial Source Code Listings" on page 311. It is recommended that you at least scan through these listings before proceeding with the tutorials.

This tutorial is limited to using VectorCAST in a single-language environment. If you will be using VectorCAST/Cover in a mixed-language environment, you will find information specific to this usage in "Using VectorCAST/Cover in a Mixed Language Environment" in the *VectorCAST/Cover User's Guide*.



Tip: You can stop a tutorial at any point and return later to where you left off. Each tutorial session is automatically saved. To return to where you left off, simply restart VectorCAST and the use **File => Recent Environments** to reopen the tutorial environment.

What You Will Accomplish

This tutorial takes you through the entire process of using VectorCAST/Cover to analyze code coverage. You will:

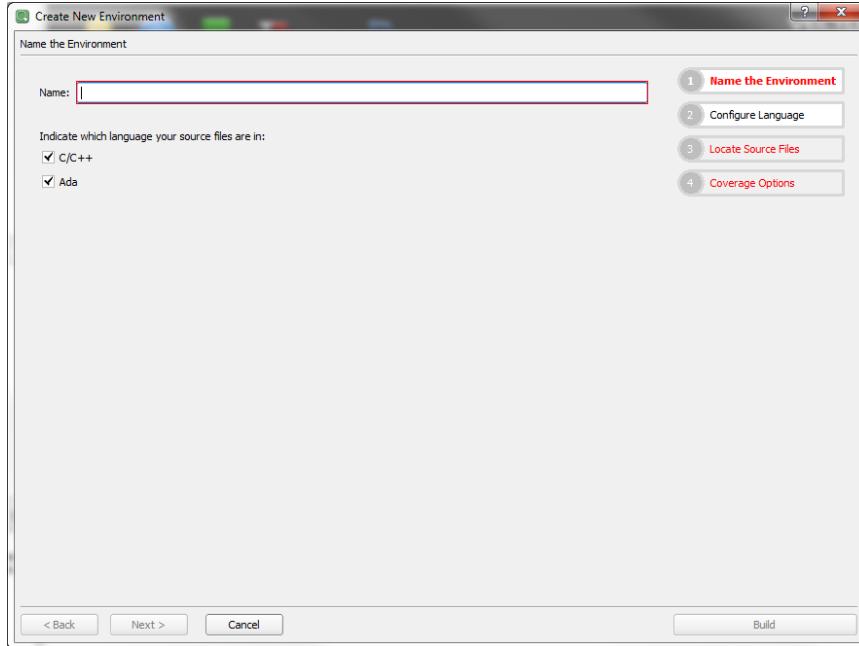
- > Build a coverage environment
- > Add the application source files to be instrumented for code coverage
- > Compile and link the application using instrumented versions of the source files
- > Execute the application to generate coverage results
- > Add results to the environment
- > View the coverage achieved with these test results
- > Add notes to the test results
- > Generate the Aggregate Coverage report for both units
- > Generate a Test Result Management report

Creating a VectorCAST/Cover Environment

In this section, you will create a coverage environment and add source files.

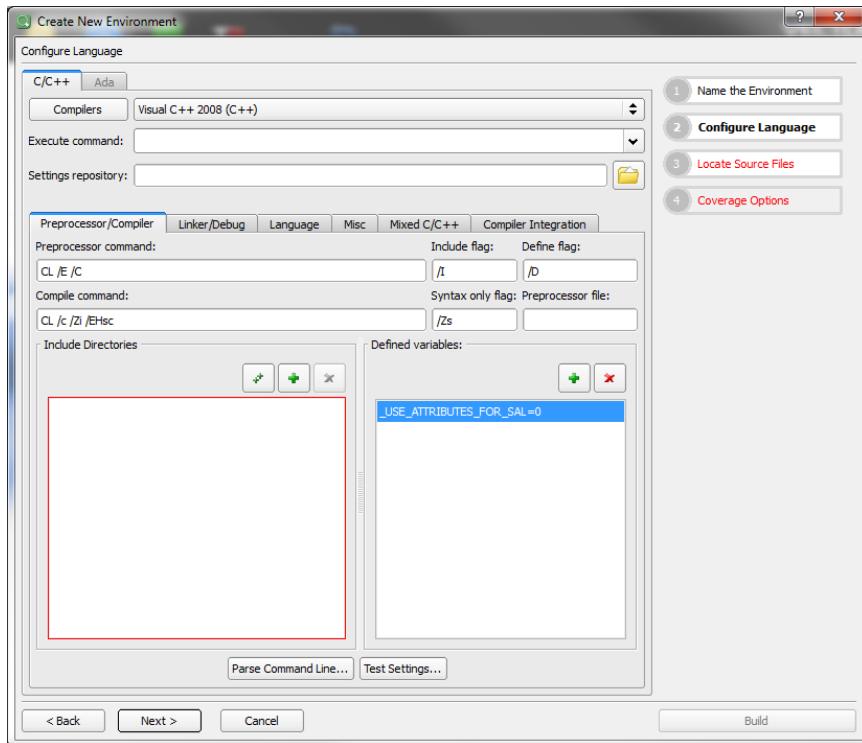
1. From the Welcome page, click the **Start Using VectorCAST** button
2. Click the link **Build a VectorCAST/Cover environment**.

The first page of the Create New Coverage Environment wizard appears:



If the former “Create Cover Environment” dialog appears instead of the new Cover Wizard, that means you have the Coverage Option “Use Legacy code instrumenter” on. It is recommended that you keep this option off unless VectorCAST Technical Support has instructed you otherwise.

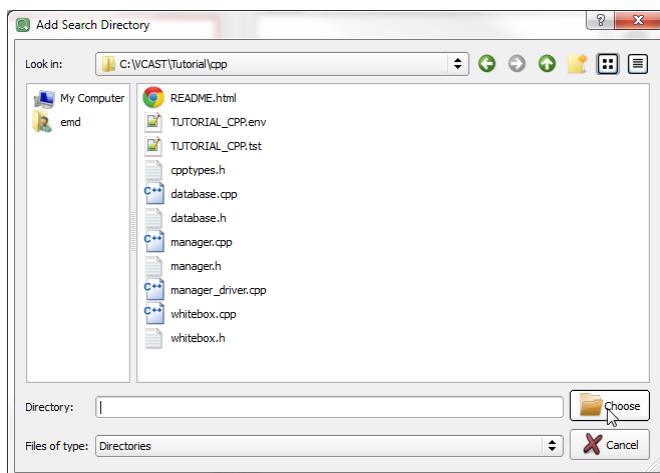
3. For the name, enter **Tutorial_cover**.
4. Uncheck **Ada**, as we are only using C++ source files in this tutorial.
5. Click **Next**.
The Configure Language page appears. Because all source files are preprocessed before instrumenting (in VectorCAST version 6.0 and above), you need to specify a compiler template.
6. Choose a compiler template. In this example, Visual Studio 2008 is selected.



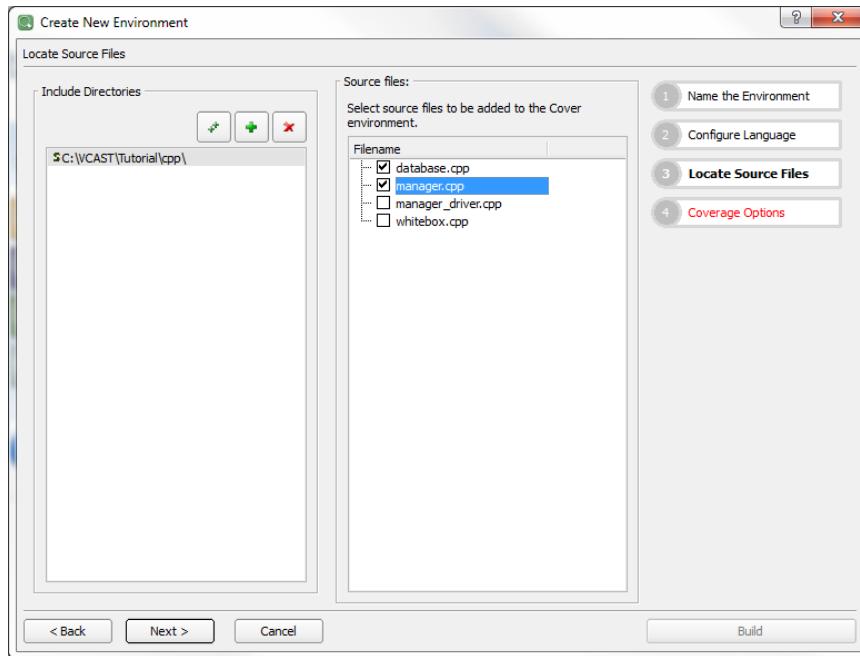
7. Click **Next**.

The Locate Source Files page appears. Here you add directories with source files that you want added to the Cover environment.

8. Click the Add Source Directory button .
9. Navigate to the **Tutorial\cpp** directory in the VectorCAST installation directory, and click Choose.

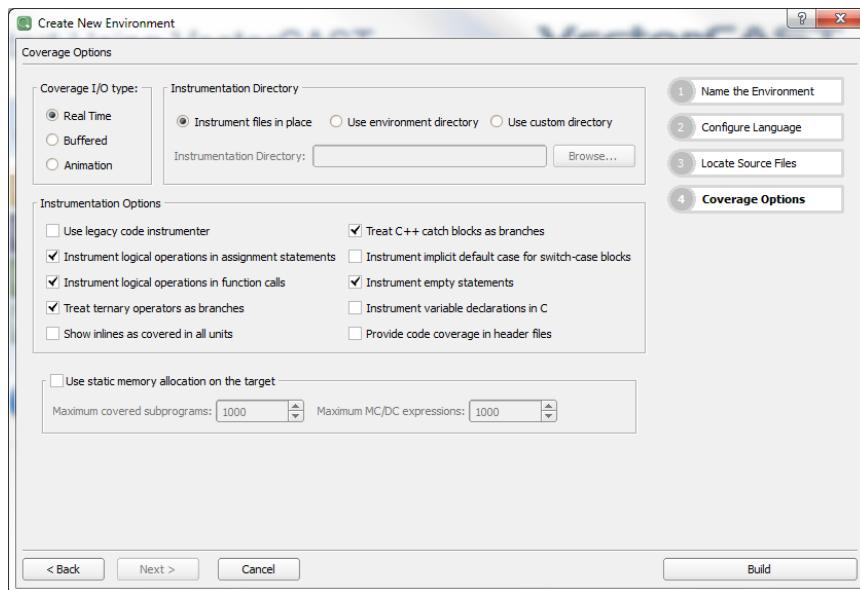


10. Put a checkmark next to **database.cpp** and **manager.cpp**.



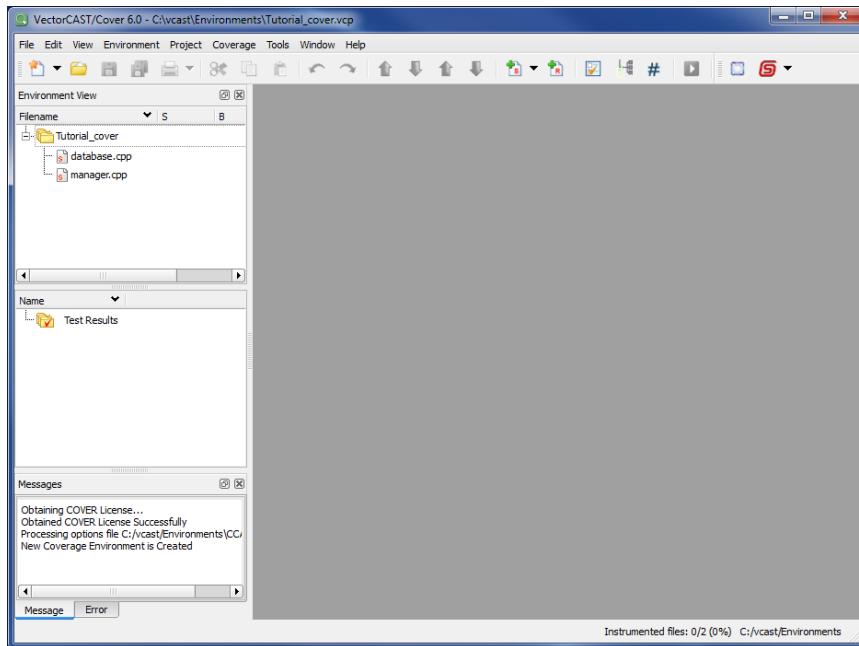
11. Click **Next >**.

The Coverage Options page appears.



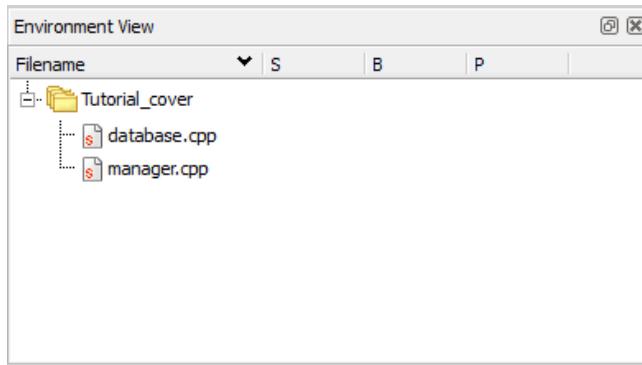
12. We don't want to change anything here, so click **Build**.

The Cover environment is created and opens.



You have now created a file named **Tutorial_cover.vcp** and a subdirectory named **Tutorial_cover**. The **.vcp** file is used to store information about the environment. The environment directory is used to store files (including code-listing and analysis files) containing instrumentation and coverage information.

The files you selected (manager.cpp and database.cpp) appear in the Source Files pane.

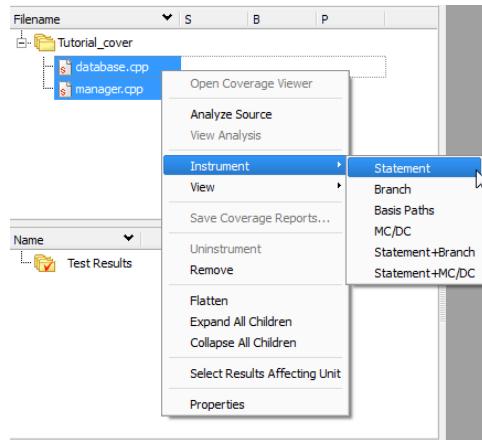


The Filename Tree contains three columns to the right of the environment and file names. These columns provide coverage status information for Statement (S), Branch (B), and MC/DC Pairs (P). These columns are currently empty because the source files have not yet been instrumented.

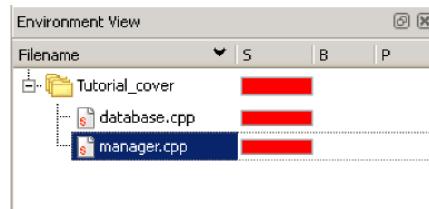
Building the Tutorial Application

In this section, you will instrument the two source files you selected for coverage:

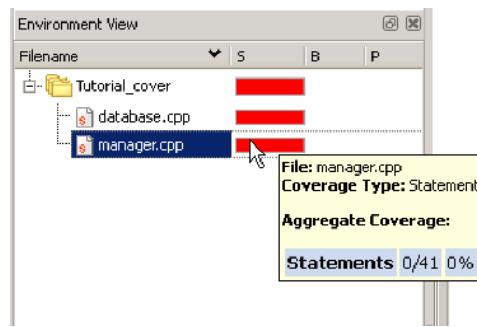
1. Right-click **database.cpp** and **manager.cpp** and select **Instrument =>Statement** from the popup menu. (If you have an Industry mode other than Default, you will see the equivalent name for Statement):



The Statement status column now contains a red status bar for each source file and for the environment as well. The status bars show the percentage of code coverage achieved for each file. For example, a file that has 50% coverage, half the bar will be green and half will be red. This allows you to quickly see a dashboard of code coverage percentages.

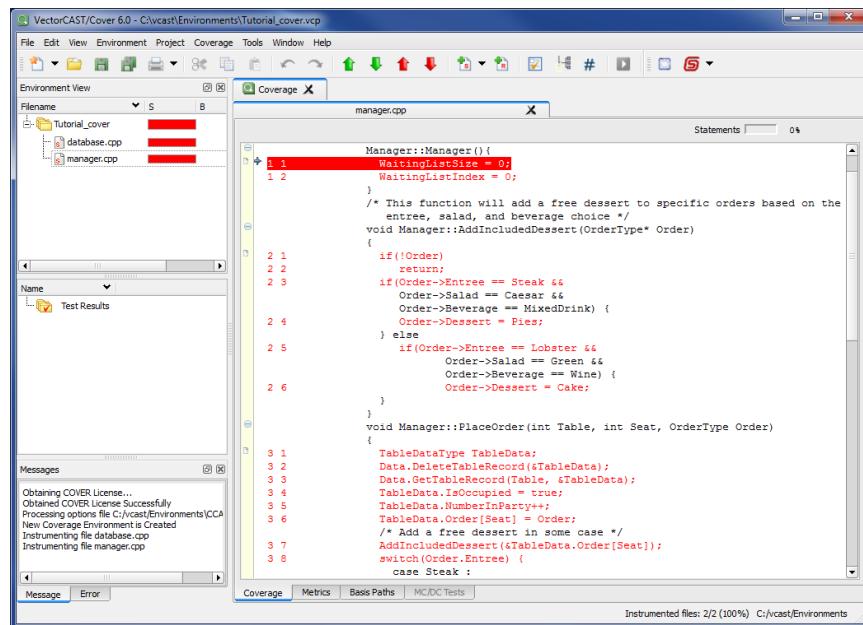


Additionally, you may hover over a file name to see the exact coverage percentages.



2. Right-click **manager.cpp** and select **Open Coverage Viewer** from the popup menu, or double-click **manager.cpp**.

The annotated source code for manager.cpp appears in the Coverage Viewer.



Each set of repeating numbers in the first column marks a subprogram within `manager.cpp`. The ascending numbers in the second column mark the executable statements within a subprogram. The statements in red have not been exercised; the statements in black cannot be exercised.

Instrumented versions of `manager.cpp` and `database.cpp` have been created in the VectorCAST installation directory `/Tutorial/cpp`. Backups of the original files are in `manager.cpp.vcast.bak` and `database.cpp.vcast.bak`.

You can now compile and link the files: `manager_driver.cpp`, `manager.cpp`, `database.cpp`, `c_cover_io.cpp` into an executable program (to be named `manager_driver`).

3. Bring up a Command Prompt and cd into the VectorCAST installation directory \Tutorial\cpp:

```
> cd %VECTORCAST_DIR%\Tutorial\cpp
C:\VCAST\Tutorial\cpp>
```

4. Using Windows Explorer or the command line, copy the coverage I/O files from the environment directory to the directory where the instrumented files reside. You can find the location of the environment by looking at the lower right side of the status bar.

```
C:\VCAST\Tutorial\cpp>copy C:\VCAST\Environments\Tutorial_cover\c_cover_io.cpp .
1 file(s) copied.
C:\VCAST\Tutorial\cpp>copy C:\VCAST\Environments\Tutorial_cover\vcast_c_options.h .
1 file(s) copied.
C:\VCAST\Tutorial\cpp>copy C:\VCAST\Environments\Tutorial_cover\c_cover.h .
1 file(s) copied.
```

5. If you are using the Visual C++ compiler (for example), enter the following compile command:



Note: If you are using the Visual C++ compiler (for example) but it is not on your PATH, enter: `C:\vcast_tutorial\CPP\Tutorial_cover>vcvars32.bat`

Typically, `vcvars32.bat` is located in the Microsoft compiler \bin directory. Optionally, you can select **Start=>Microsoft Visual Studio=>Visual Studio Tools=>Visual Studio Command Prompt**. This will open a command prompt window with `vcvars32.bat` already invoked.

```
C:\VCAST\Tutorial\cpp>cl /Femanager_driver manager.cpp database.cpp manager_
driver.cpp c_cover_io.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.30729.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

manager.cpp
database.cpp
manager_driver.cpp
c_cover_io.cpp
Generating Code...
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.
/out:manager_driver.exe
manager.obj
database.obj
manager_driver.obj

c_cover_io.obj
```

If you are using the GNU Native compiler (for example), enter the following compile command:

```
/vcast_tutorial/CPP/Tutorial_cover>g++ -o manager_driver manager.cpp database.cpp manager_
driver.cpp c_cover_io.cpp
```

You can now execute **manager_driver** to generate coverage results.

Executing the Instrumented Program

You will execute `manager_driver` four times, once for each of the four possible input values (P, C, G, A). Each invocation creates a results file, `TESTINSS.DAT`, which you will copy to a file of another name.

1. In a DOS or shell window, enter (DOS format):

```
C:\VCAST\Tutorial\CPP>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert :
```

Each value (P, C, G, A) invokes a different subprogram defined in `manager.cpp`.

2. Enter **P** to invoke the subprogram `Place_Order`, and then press **Enter**.

```
C:\VCAST\Tutorial\CPP>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : P
```

Coverage data is written to a file named `TESTINSS.DAT` in directory `C:\VCAST\Tutorial\CPP`.

3. Copy the `TESTINSS.DAT` file to `Place_Order_Results.DAT`.

```
C:\VCAST\Tutorial\CPP>copy TESTINSS.DAT Place_Order_Results.DAT
```

4. Continue with C:

```
C:\VCAST\Tutorial\CPP>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : C
C:\VCAST\Tutorial\CPP>copy TESTINSS.DAT Clear_Table_Results.DAT
```

5. and G:

```
C:\VCAST\Tutorial\CPP>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : G
The Total is 10
C:\VCAST\Tutorial\CPP>copy TESTINSS.DAT Get_Check_Total.DAT
```

6. and finally A:

```
C:\VCAST\Tutorial\CPP>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : A
C:\VCAST\Tutorial\CPP>copy TESTINSS.DAT Add_Included_Dessert.DAT
```

7. Remove TESTINSS.DAT.

```
C:\VCAST\Tutorial\CPP>del TESTINSS.DAT
```

Adding Test Results

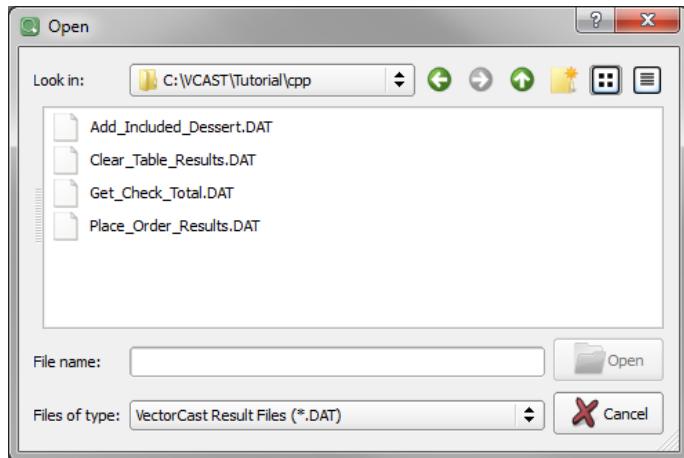
In order to view these coverage results, you must first add them to the environment.

1. If necessary, restart VectorCAST, and use **File => Recent Environments** to re-open the coverage environment.
2. In VectorCAST/Cover, select **Environment => Add Test Results...** from the main-menu bar, or click  on the toolbar, or drag a results file from outside VectorCAST into the Test Results pane.

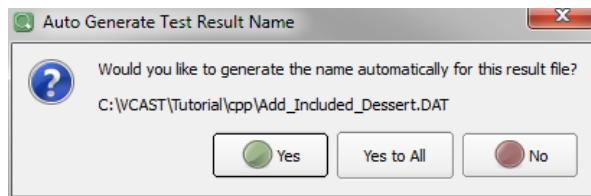


Note: Do not use **Coverage => Import Results from Environment**. This command is used to import coverage data from another VectorCAST environment.

3. In the Open dialog, navigate to the **Tutorial\cpp** directory, and multi-select the four .DAT files and then click **Open**.

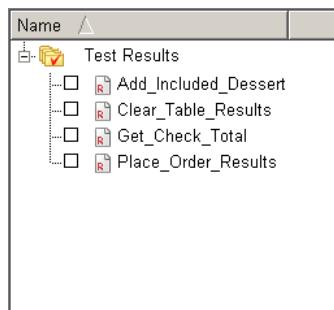


You are asked if you want to generate an automatic name for the test results.



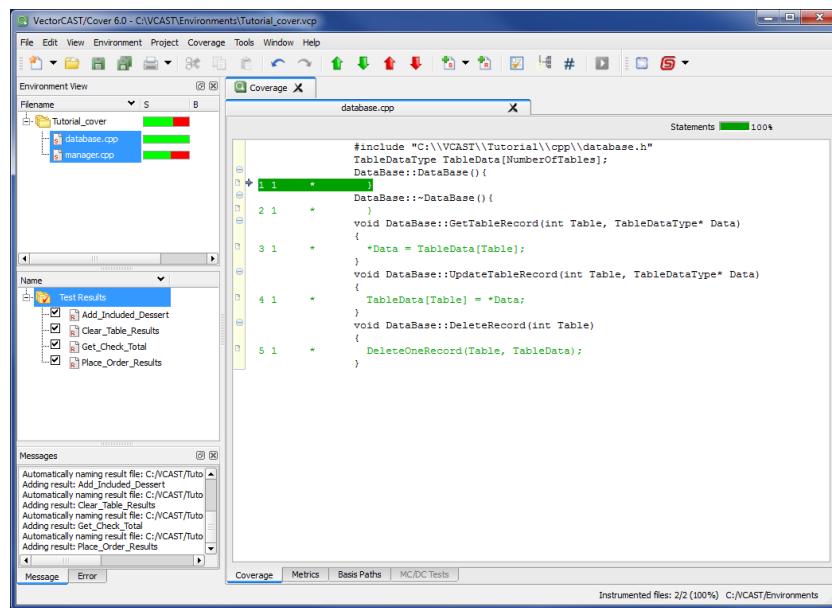
4. Click **Yes to All**.

VectorCAST adds each test result file, naming it the same name as its base filename.



5. Right-click **Test Results**, and select **Select All Children** from the popup menu.

A checkmark appears in front of each test result, and the tab for database.cpp opens in the Coverage Viewer, because the constructor DataBase::DataBase() is the first line called when the program is executed.

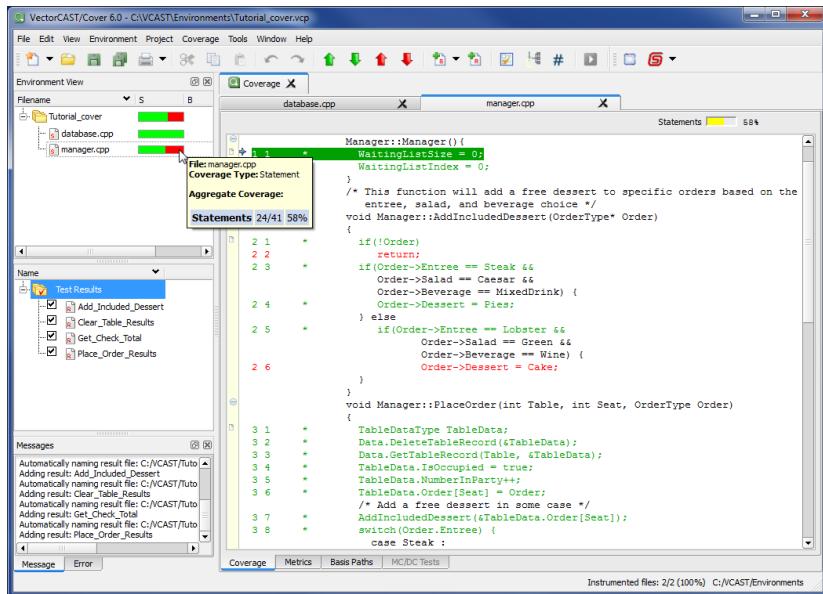


Green indicates lines that were exercised when we invoked **manager_driver**; red indicates unexecuted lines; black indicates non-executable statements.

The status bar at the upper-right corner of the Coverage Viewer tells you that 100% of the executable statements in the unit database.cpp have been covered.

The status bars in the Filename Tree reflect percentage of coverage as well. Database.cpp has 100% statement coverage.

6. Double-click **manager.cpp** to open it in the Coverage Viewer. The status bar shows that 58% of the executable statements in the unit manager.cpp have been covered. By hovering over the status bar in the Filename Tree for manager.cpp, you can easily see that 24 of 41 statements were covered.

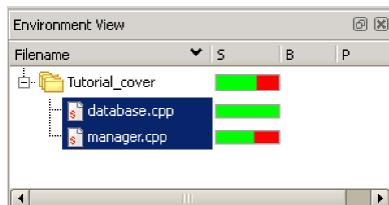


Note: VectorCAST's animation feature allows you to view the step-by-step coverage of a test case. This feature is described in the user guide accompanying your version of VectorCAST.

View the Aggregate Coverage Report

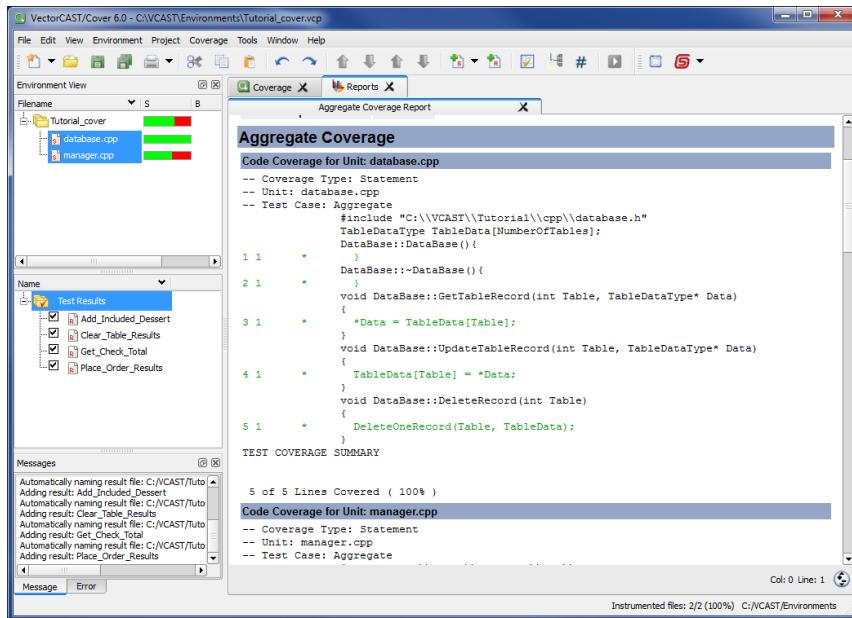
The Aggregate Coverage report includes the annotated source code and Metrics table for the selected units, with all Test Results turned on. In this section, you will generate an Aggregate Coverage report for both units in the environment.

1. Select (highlight) both units in the Source Files pane or the Tutorial_cover environment folder.



2. Choose **Environment => View => Aggregate Coverage Report**.

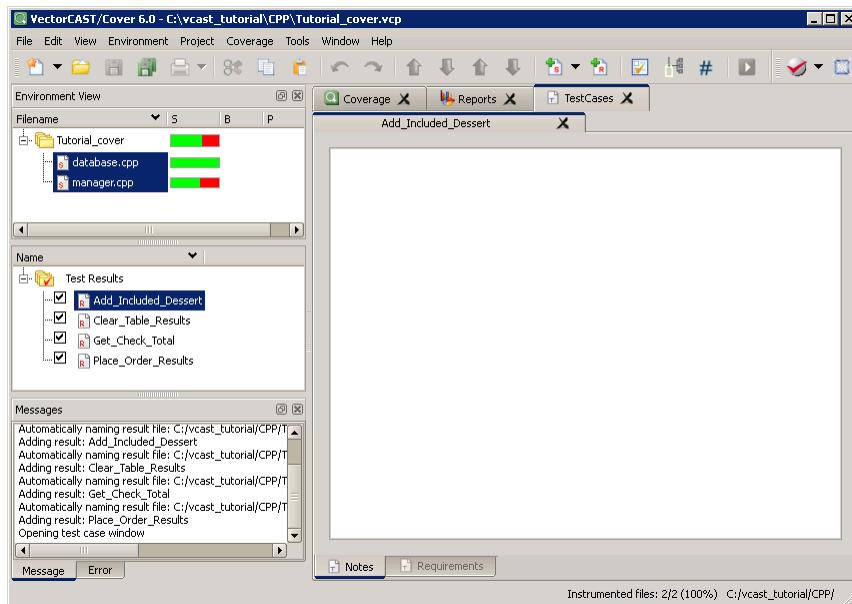
The Aggregate Coverage report is displayed in the Report Viewer:



View the Management Report

The Management report includes a summary of Test Results and Metrics table for the selected units, with all Test Results turned on. In this section, you will generate a Management report for both units in the environment.

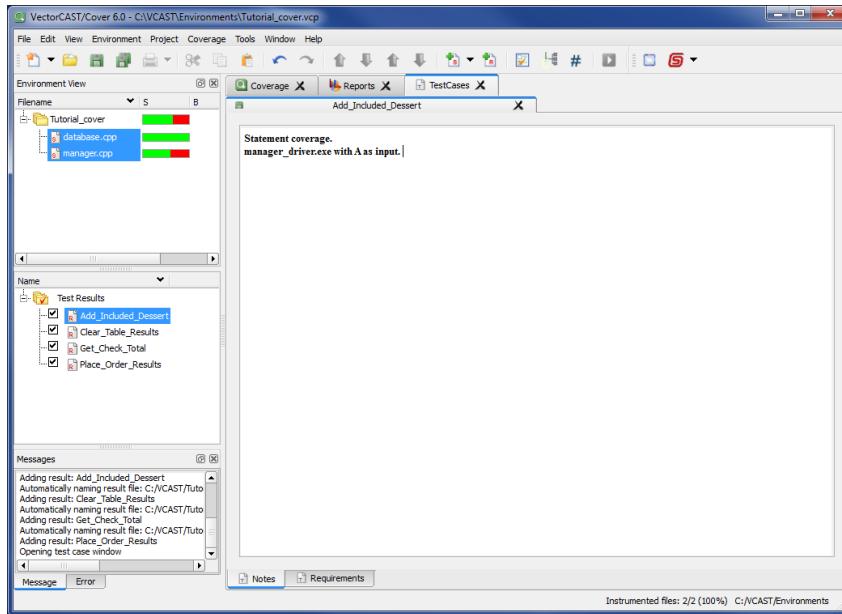
1. Double-click the Test Result named **Add_Included_Dessert**.
A viewer opens, displaying an empty Notes tab for the Test Result.



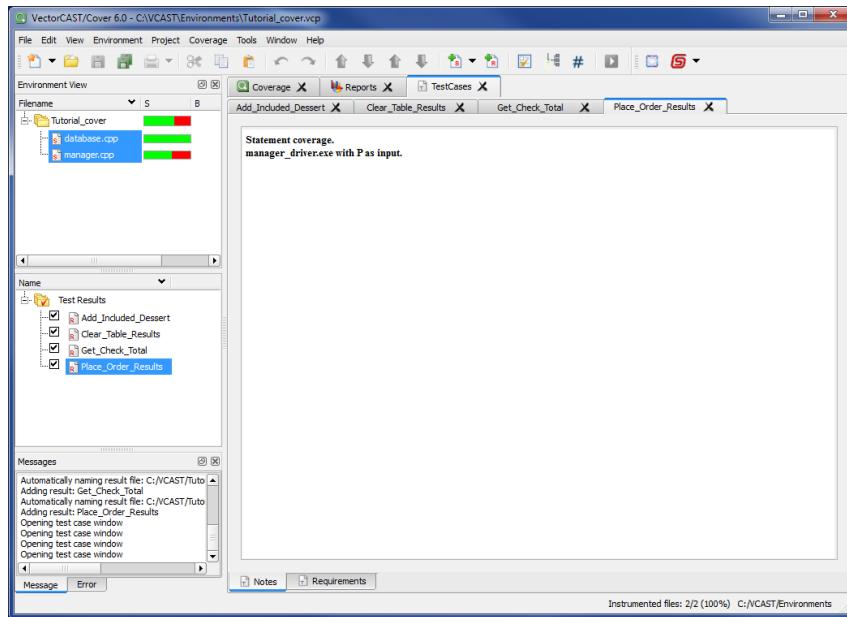


Note: An additional “Requirements” tab is included for environments associated with a Requirements Gateway repository.

2. In the Notes tab, enter text of your choices, such as, “**Statement coverage. manager_driver.exe with A as input.**”
3. Click **Save**.



4. Repeat for the other Test Results, changing the text to reflect the input that was used for that Test Result.



5. Select (highlight) both units in the Source Files pane or the Tutorial_cover folder.



6. Choose **Environment => View => Management Report**.

The Test Results Management report is displayed in the Report Viewer.

7. Click the link **Test Results Management** or just scroll down a bit.

Test Results Management		
Test Results	Addition Time	Notes
Add_Included_Dessert	4 Jun 2012 11:21:51 AM	Statement coverage. manager_driver.exe with A as input.
Clear_Table_Results	4 Jun 2012 11:21:51 AM	Statement coverage. manager_driver.exe with C as input.
Get_Check_Total	4 Jun 2012 11:21:51 AM	Statement coverage. manager_driver.exe with G as input.
Place_Order_Results	4 Jun 2012 11:21:51 AM	Statement coverage. manager_driver.exe with P as input.

Metrics			
Unit	Subprogram	Complexity	Statements
database.cpp	DataBase:: DataBase	1	100% (1 / 1)
	DataBase::~ DataBase	1	100% (1 / 1)
	DataBase::GetTableRecord	1	100% (1 / 1)
	DataBase::UpdateTableRecord	1	100% (1 / 1)
	DataBase::DeleteRecord	1	100% (1 / 1)
TOTALS	5	5	100% (5 / 5)
manager.cpp	Manager:: Manager	1	100% (2 / 2)
	Manager::AddIncludedDessert	4	66% (4 / 6)
	Manager::PlaceOrder	5	72% (13 / 18)
	Manager::ClearTable	1	100% (1 / 1)

You see that this part of the report includes the name of the Test Result, the date and time it was added to the environment, and the notes associated with each.

Tutorial Summary

VectorCAST/Cover is an effective tool for analyzing test coverage, and for determining what additional testing needs to be done in order to fully exercise an application.

This tutorial took you through the entire process of using VectorCAST/Cover to analyze statement code coverage.

In this tutorial, you:

- > Built a coverage environment
- > Added the application source files to be instrumented for code coverage
- > Compiled and linked the application using instrumented versions of the source files
- > Generated test results
- > Viewed the coverage achieved with these test results
- > Added notes to the test results
- > Generated an Aggregate Coverage report for both units
- > Generated a Test Result Management report

Ada Cover Tutorial

This tutorial steps you through the full process of using VectorCAST/Cover to analyze code coverage. You will create a VectorCAST/Cover environment, instrument source files, and compile a sample application.

The source modules you will use in this tutorial are components of a simple order-management

application for restaurants. The listings for this application are available in Appendix A, "Tutorial Source Code Listings" on page 311. It is recommended that you at least scan through these listings before proceeding with the tutorials.

This tutorial is limited to using VectorCAST in a single-language environment. If you will be using VectorCAST/Cover in a mixed-language environment, you will find information specific to this usage in "Using VectorCAST/Cover in a Mixed Language Environment" in the *VectorCAST/Cover User's Guide*.



Tip: You can stop a tutorial at any point and return later to where you left off. Each tutorial session is automatically saved. To return to where you left off, simply restart VectorCAST and the use **File => Recent Environments** to reopen the tutorial environment.

What You Will Accomplish

This tutorial takes you through the entire process of using VectorCAST/Cover to analyze code coverage. You will:

- > Build a coverage environment
- > Add the application source files to be instrumented for code coverage
- > Compile and link the application using instrumented versions of the source files
- > Execute the application to generate coverage results
- > Add results to the environment
- > View the coverage achieved with these test results
- > Add notes to the test results
- > Generate the Aggregate Coverage report for both units
- > Generate a Test Result Management report

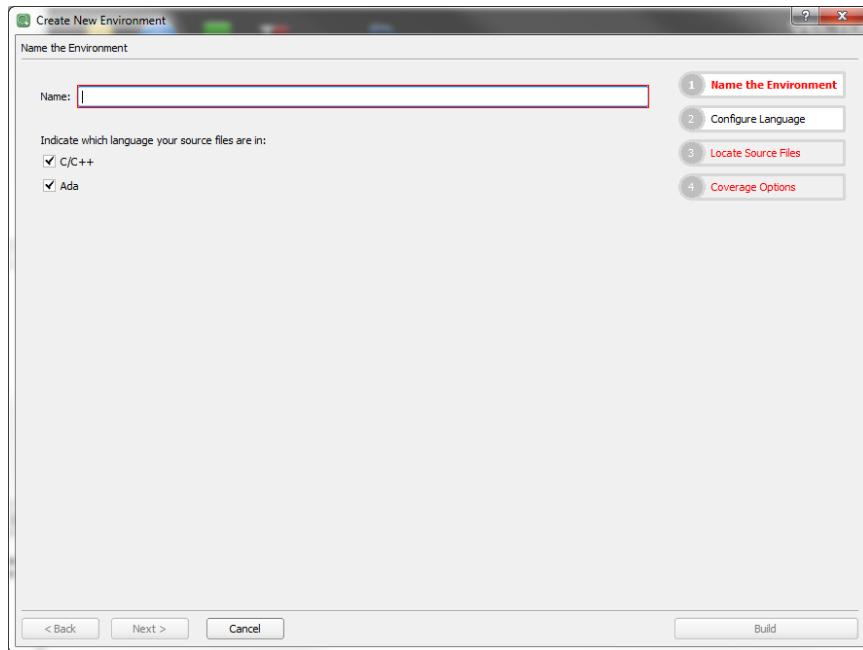
Creating a VectorCAST/Cover Environment

In this section, you will create a coverage environment and add source files.



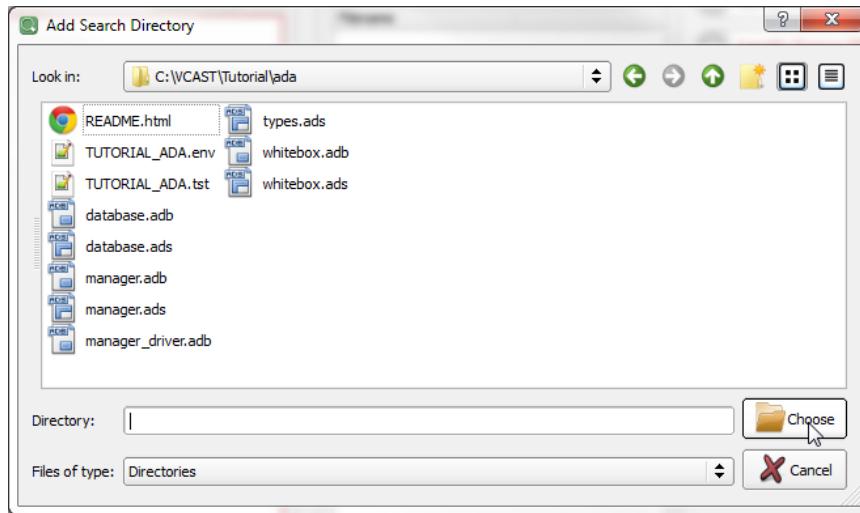
1. From the Welcome page, click the **Start Using VectorCAST** button .
2. Click the link **Build a VectorCAST/Cover environment**.

The first page of the Create New Coverage Environment wizard appears:

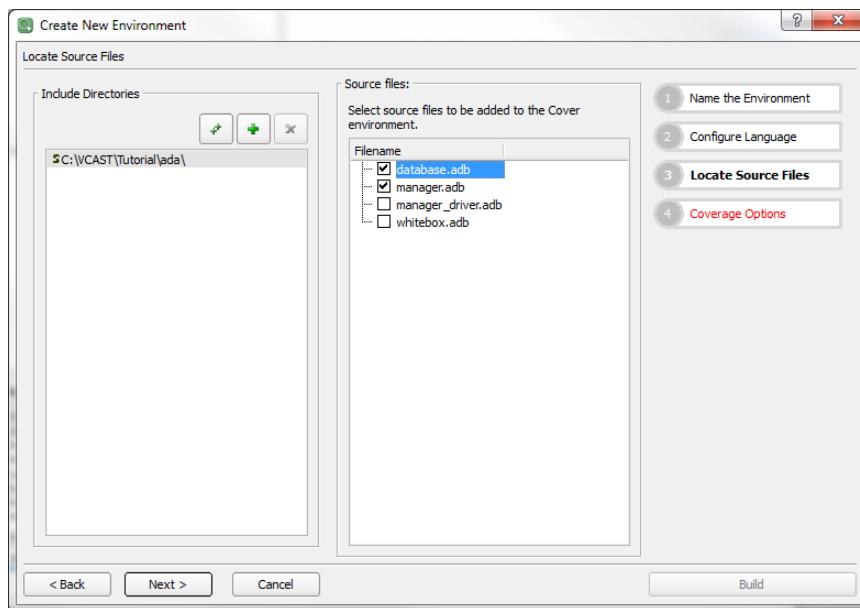


If the former “Create Cover Environment” dialog appears instead of the new Cover Wizard, that means you have the Coverage Option “Use Legacy code instrumenter” on. It is recommended that you keep this option off unless VectorCAST Technical Support has instructed you otherwise.

3. For the name, enter **Tutorial_cover**.
4. Uncheck **C++**, as we are only using Ada source files in this tutorial.
5. Click **Next**.
The Configure Language page appears. For Ada, this page is provided so you can add source file extensions if you are using any other than those listed.
6. Click **Next**.
The Locate Source Files page appears. Here you add directories with source files that you want added to the Cover environment.
7. Click the Add Source Directory button
8. Navigate to the **Tutorial\ada** directory in the VectorCAST installation directory, and click Choose.

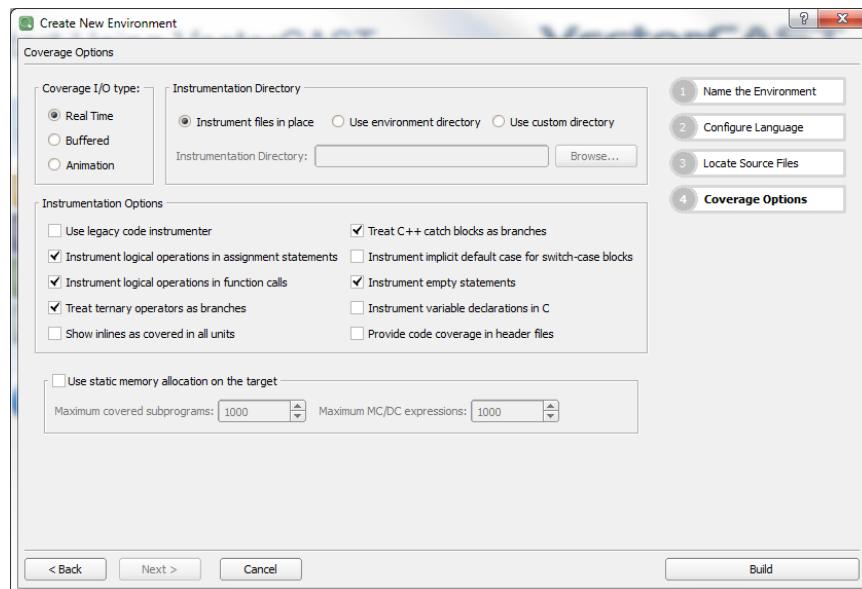


9. Put a checkmark next to **database.adb** and **manager.adb**.

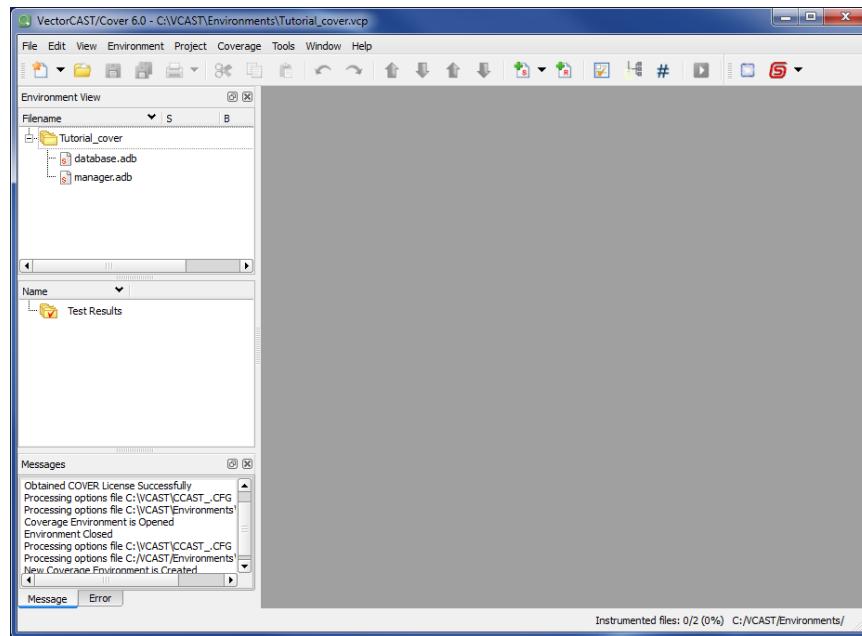


10. Click **Next**.

The Coverage Options page appears.

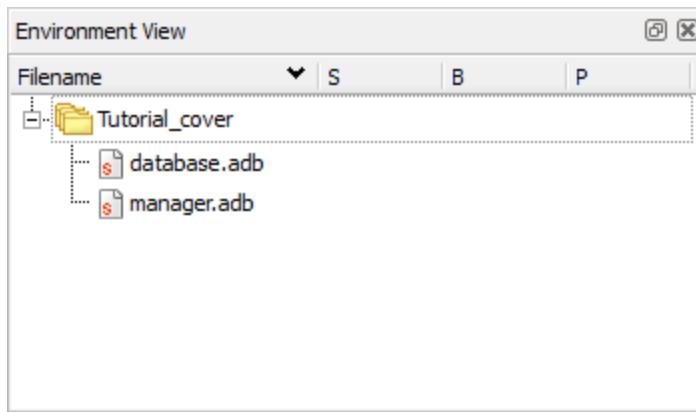


11. We don't want to change anything here, so click **Build**.
The Cover environment is created and opens.



You have now created a file named `Tutorial_cover.vcp` and a subdirectory named `Tutorial_cover`. The `.vcp` file is used to store information about the environment. The environment directory is used to store files (including code-listing and analysis files) containing instrumentation and coverage information.

The files you selected (`manager.adb` and `database.adb`) appear in the Source Files pane.

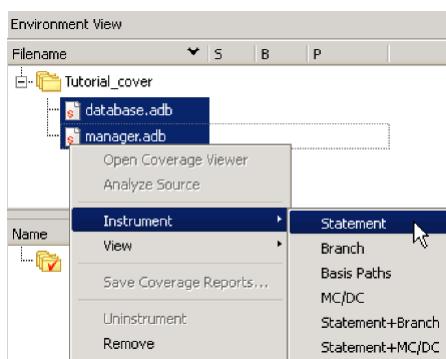


The Filename Tree contains three columns to the right of the environment and file names. These columns provide coverage status information for Statement (S), Branch (B), and MC/DC Pairs (P). These columns are currently empty because the source files have not yet been instrumented.

Building the Tutorial Application

In this section, you will instrument the two source files you selected for coverage:

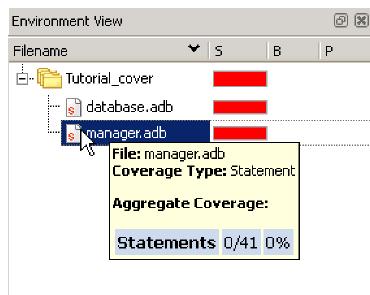
1. Right-click `database.adb` and `manager.adb` select **Instrument => Statement** from the popup menu. (If you have an Industry mode other than Default, you will see the equivalent name for Statement):



The Statement status column now contains a red status bar for each source file and for the environment as well. The status bars show the percentage of code coverage achieved for each file. For example, a file that has 50% coverage, half the bar will be green and half will be red. This allows you to quickly see a dashboard of code coverage percentages.



Additionally, you may hover over a file name to see the exact coverage percentages.



- Right-click **manager.adb** and select **Open Coverage Viewer** from the popup menu, or double-click **manager.adb**.

The annotated source code in manager.adb appears in the Coverage Viewer:

The screenshot shows the 'Coverage Viewer' window for 'manager.adb'. The left pane shows the environment view with 'manager.adb' selected. The right pane displays the annotated source code. Annotations are shown as red highlights on specific lines of code, indicating coverage status. The code is as follows:

```

1 1     DATABASE.GET_TABLE_RECORD (
2         TABLE => TABLE,
3         DATA => TABLE_DATA);
4
5         TABLE_DATA.IS_OCCUPIED := true;
6         TABLE_DATA.NUMBER_IN_PARTY := TABLE_DATA.NUMBER_IN_PARTY;
7         TABLE_DATA.ORDER (SEAT) := ORDER;
8
9         ADD_INCLUDED_DESSERT ( TABLE_DATA.ORDER(SEAT) );
10
11        case ORDER_ENTREE is
12            when TYPES.NO_ORDER =>
13                null;
14            when TYPES.STEAK =>
15                TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL +
16                TABLE_DATA.CHICKEN =>
17                    TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL +
18                    TABLE_DATA.LOBSTER =>
19                        TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL +
20                        TABLE_DATA.PASTA =>
21                            TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL +
22                            end case;
23
24        DATABASE.UPDATE_TABLE_RECORD (
25            TABLE => TABLE,
26            DATA => TABLE_DATA);

```

Each set of repeating numbers in the first column marks a subprogram within manager.adb. The ascending numbers in the second column mark the executable statements within a subprogram. The statements in red have not been exercised; the statements in black cannot be exercised.

Instrumented versions of manager.adb and database.adb have been created in the VectorCAST `installation directory\Tutorial\ada`. Backups of the original source files are in manager.adb.vcast.bak and database.adb.vcast.bak.

You can now compile and link the files manager_driver.cpp, manager.cpp, database.cpp, c_cover_io.cpp into an executable program (to be named manager_driver).

3. Bring up a Command Prompt and cd into the VectorCAST installation directory\Tutorial\ada:

```
> cd %VECTORCAST_DIR%\Tutorial\ada
C:\VCAST\Tutorial\ada>
```

4. If you are using a GNAT compiler, before you can compile the instrumented files, you must split each file into a types file and a residual instrumented file. To do this, you run the **gnatchop** command on each instrumented file. For example:

```
C:\vcast_tutorial\ADA>gnatchop -w manager.adb database.adb
splitting manager.adb into:
  vcast_types_2.ads
  manager.adb
splitting database.adb into:
  vcast_types_2.ads
  database.adb
```

5. Using Windows Explorer or the command line, copy the coverage I/O files from the environment directory to the directory where the instrumented files reside. You can find the location of the environment by looking at the lower right side of the status bar.

```
C:\VCAST\Tutorial\ada>copy C:\VCAST\Environments\Tutorial_cover\vcast_cover_io.adan>
  2 file(s) copied.
C:\VCAST\Tutorial\ada>copy C:\VCAST\Environments\Tutorial_cover\vcast_ada_
options.ad*.
  2 file(s) copied.
```

You can now compile and link the files manager_driver.adb, manager.adb, database.adb, and vcast_cover_io.adbn> into an executable program (to be named **manager_driver**).

6. If you are using a GNAT compiler (for example), enter:

```
C:\vcast_tutorial\ADA>gnatmake manager_driver
gcc -c manager_driver.adb
gcc -c manager.adb
gcc -c types.ads
gcc -c database.adb
gcc -c vcast_cover_io.adb
gcc -c vcast_types_2.ads
gcc -c vcast_types_1.ads
gcc -c vcast_ada_options.adb
gnatbind -x manager_driver.ali
gnatlink manager_driver.ali
```

You can now execute **manager_driver** to generate coverage results.

Executing the Instrumented Program

You will execute **manager_driver** four times, once for each of the four possible input values (P, C, G, A). Each invocation creates a results file, **TESTINSS.DAT**, which you will copy to a file of another name.

1. In a DOS or shell window, enter (DOS format):

```
C:\VCAST\Tutorial\ada>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert :
```

Each value (P, C, G, A) invokes a different subprogram defined in **manager.adb**.

2. Enter **P** to invoke the subprogram **Place_Order**, and then press **Enter**.

```
C:\VCAST\Tutorial\ada>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : P
```

Coverage data is written to a file named **TESTINSS.DAT** in directory **C:\VCAST\Tutorial\ada**.

3. Copy the **TESTINSS.DAT** file to **Place_Order_Results.DAT**.

```
C:\VCAST\Tutorial\ada>copy TESTINSS.DAT Place_Order_Results.DAT
```

4. Continue with **C**:

```
C:\VCAST\Tutorial\ada>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : C
C:\VCAST\Tutorial\ada>copy TESTINSS.DAT Clear_Table_Results.DAT
```

5. and **G**:

```
C:\VCAST\Tutorial\ada>manager_driver
P=PlaceOrder  C=ClearTable  G=GetCheckTotal  A=AddIncludedDessert : G
The Total is 10
C:\VCAST\Tutorial\ada>copy TESTINSS.DAT Get_Check_Total.DAT
```

6. and finally **A**:

```
C:\VCAST\Tutorial\ada>manager_driver
P=PlaceOrder C=ClearTable G=GetCheckTotal A=AddIncludedDessert : A
C:\VCAST\Tutorial\ada>copy TESTINSS.DAT Add_Included_Dessert.DAT
```

7. Remove TESTINSS.DAT.

```
C:\VCAST\Tutorial\ada>del TESTINSS.DAT
```

Adding Test Results

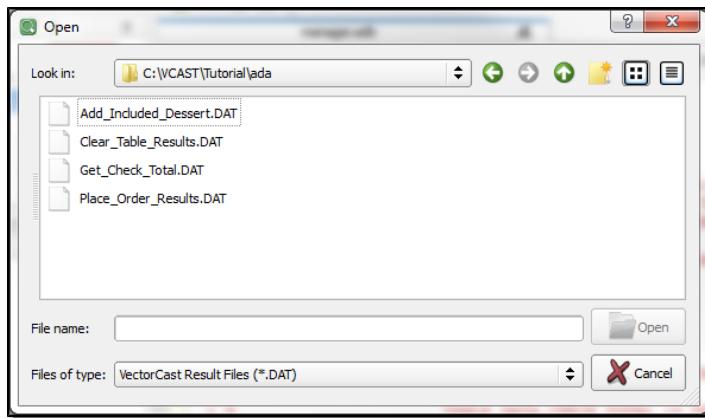
In order to view these coverage results, you must first add them to the environment.

1. If necessary, restart VectorCAST, and use **File => Recent Environments** to re-open the coverage environment.
2. In VectorCAST/Cover, select **Environment => Add Test Results...** from the main-menu bar, or click  on the toolbar, or drag a results file from outside VectorCAST into the Test Results pane.

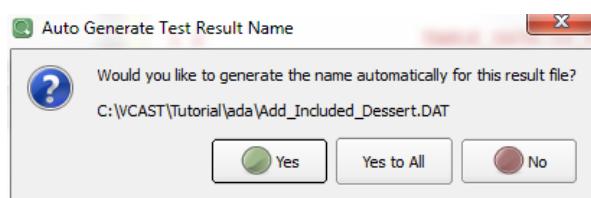


Note: Do not use **Coverage => Import Results from Environment**. This command is used to import coverage data from another VectorCAST environment.

3. In the Open dialog, navigate to the **Tutorial\ada** directory, and multi-select the four .DAT files and then click **Open**.

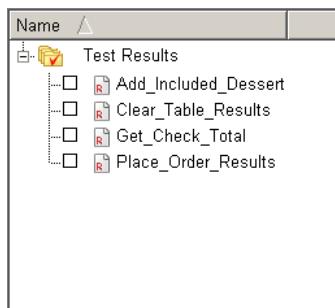


You are asked if you want to generate an automatic name for the test results.



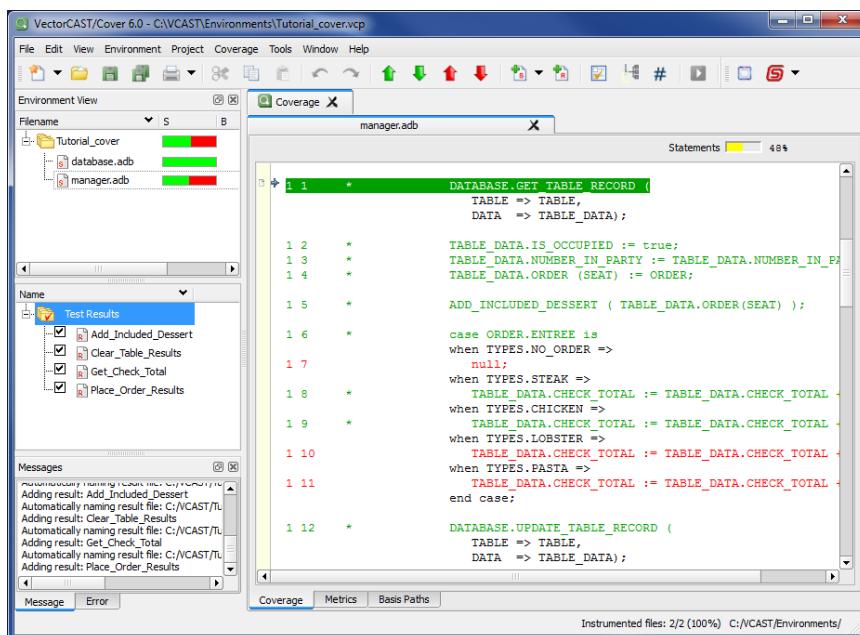
4. Click **Yes to All**.

VectorCAST adds each test result file, naming it the same name as its base filename.



5. Right-click **Test Results**, and select **Select All Children** from the popup menu.

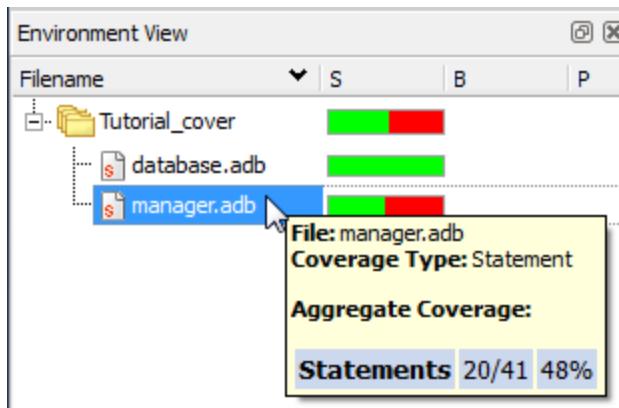
A checkmark appears in front of each test results, and the tab for manager.adb opens in the Coverage Viewer.



Green indicates lines that were exercised when we invoked `manager_driver`; red indicates unexecuted lines; black indicates non-executable statements.

6. The status bar at the upper-right corner of the Coverage Viewer tells you that 48% of the executable statements in the unit manager.adb have been covered.

The status bars in the Filename Tree reflect percentage of coverage as well. Manager.adb has 48% statement coverage. By hovering over the status bar in the Filename Tree for manager.adb, you can easily see that 20 of 41 statements were covered.

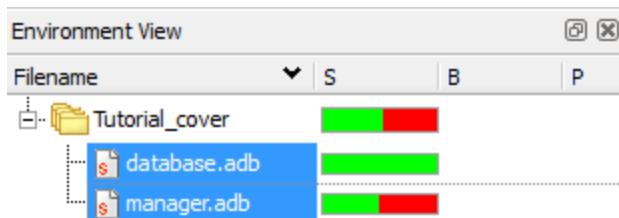


Note: VectorCAST's animation feature allows you to view the step-by-step coverage of a test case. This feature is described in the user guide accompanying your version of VectorCAST.

View the Aggregate Coverage Report

The Aggregate Coverage report includes the annotated source code and Metrics table for the selected units, with all Test Results turned on. In this section, you will generate an Aggregate Coverage report for both units in the environment.

1. Select (highlight) both units in the Source Files pane or the Tutorial_cover environment folder.



2. Choose **Environment => View => Aggregate Coverage Report**.

The Aggregate Coverage report is displayed in the Report Viewer:

```

Aggregate Coverage
Code Coverage for Unit: database.adb
-- Coverage Type: Statement
-- Unit: database.adb
-- Test Case: Aggregate
with types;

package body DATABASE is

    TABLE_DATA : array (TYPES.TABLE_INDEX_TYPE) of TYPES.TABLE_D

    procedure GET_TABLE_RECORD (
        TABLE : in     TYPES.TABLE_INDEX_TYPE;
        DATA : out TYPES.TABLE_DATA_TYPE) is
begin
    DATA := TABLE_DATA (TABLE);
end GET_TABLE RECORD;

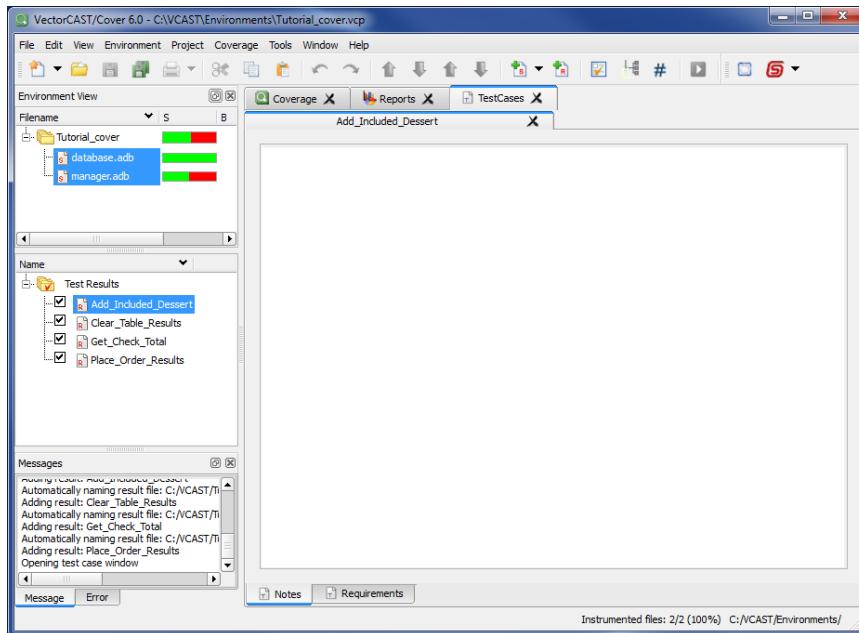
procedure UPDATE_TABLE_RECORD (
    TABLE : in     TYPES.TABLE_INDEX_TYPE;
    DATA : in      TYPES.TABLE_DATA_TYPE) is
begin
    TABLE_DATA (TABLE) := DATA;
end UPDATE_TABLE_RECORD;

```

View the Management Report

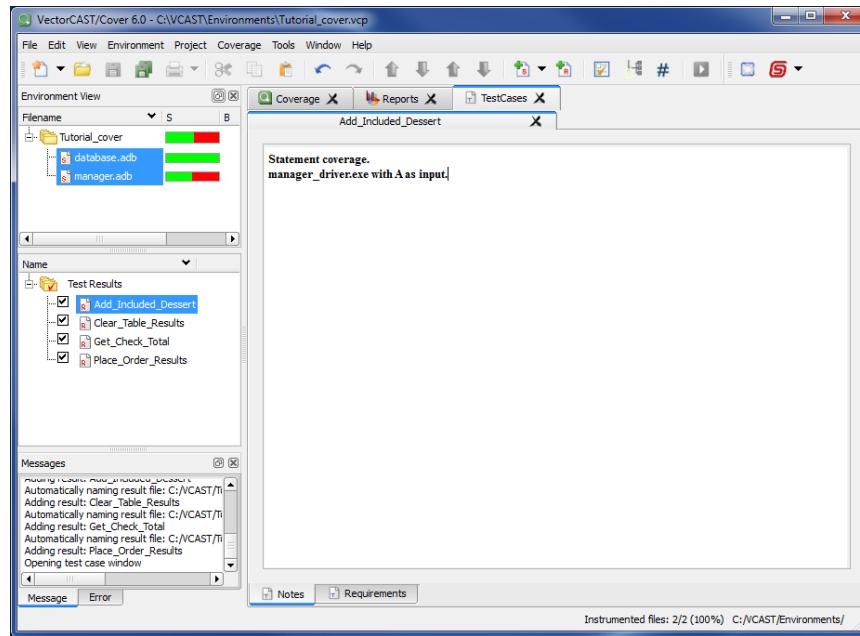
The Management report includes a summary of Test Results and Metrics table for the selected units, with all Test Results turned on. In this section, you will generate a Management report for both units in the environment.

1. Double-click the Test Result named **Add_Included_Dessert**.
A viewer opens, displaying an empty Notes tab for the Test Result.

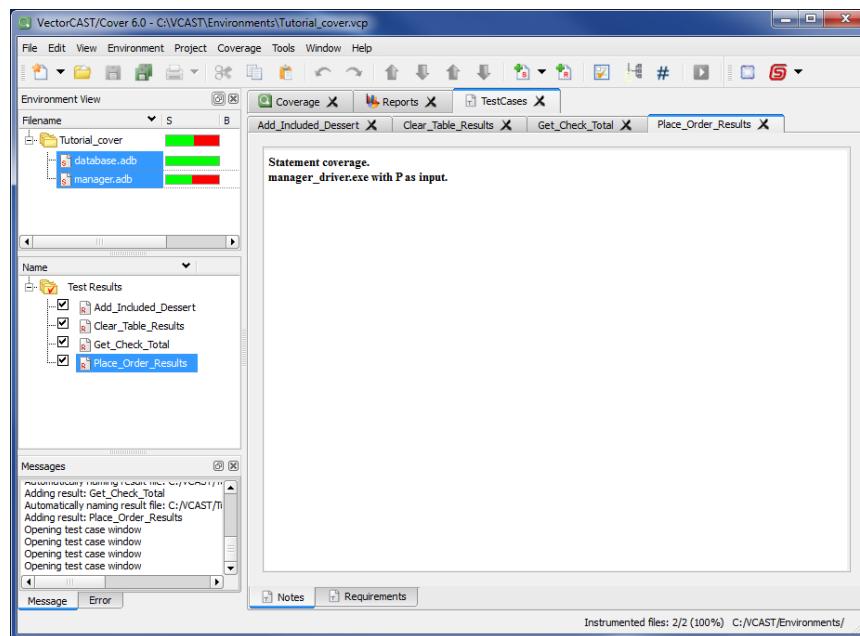


Note: An additional “Requirements” tab is included for environments associated with a Requirements Gateway repository.

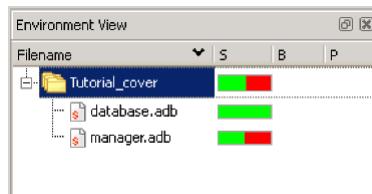
2. In the Notes tab, enter text of your choices, such as, “**Statement coverage. manager_driver.exe with A as input.**”
3. Click **Save**.



4. Repeat for the other Test Results, changing the text to reflect the input that was used for that Test Result.



5. Select (highlight) both units in the Source Files pane or the Tutorial_cover folder.



6. Choose **Environment => View => Management Report**.
The Test Results Management report is displayed in the Report Viewer.
7. Click the link **Test Results Management** or just scroll down a bit.

Aggregate Coverage Report **X** **Test Results Management Report** **X**

Test Results Management

Test Results	Addition Time	Notes
Add_Included_Dessert	4 Jun 2012 2:08:44 PM	Statement coverage. manager_driver.exe with A as input.
Clear_Table_Results	4 Jun 2012 2:08:45 PM	Statement coverage. manager_driver.exe with C as input.
Get_Check_Total	4 Jun 2012 2:08:45 PM	Statement coverage. manager_driver.exe with G as input.
Place_Order_Results	4 Jun 2012 2:08:45 PM	Statement coverage. manager_driver.exe with P as input.

Metrics

Unit	Subprogram	Complexity	Statements
database.adb	GET_TABLE_RECORD	1	100% (1 / 1)
	UPDATE_TABLE_RECORD	1	100% (1 / 1)
	TOTALS	2	100% (2 / 2)
manager.adb	PLACE_ORDER	5	75% (9 / 12)
	ADD_INCLUDED_DESSERT	3	66% (2 / 3)
	CLEAR_TABLE	2	100% (7 / 7)
	GET_CHECK_TOTAL	1	100% (2 / 2)
	ADD_PARTY_TO_WAITING_LIST	2	0% (0 / 5)
	GET_NEXT_PARTY_TO_BE_SEATED	6	0% (0 / 12)
	TOTALS	6	48% (20 / 41)

Col: 0 Line: 1

Tutorial Summary

VectorCAST/Cover is an effective tool for analyzing test coverage, and for determining what additional testing needs to be done in order to fully exercise an application.

This tutorial took you through the entire process of using VectorCAST/Cover to analyze statement code coverage.

In this tutorial, you:

- > Built a coverage environment
- > Added the application source files to be instrumented for code coverage

- > Compiled and linked the application using instrumented versions of the source files
- > Generated test results
- > Viewed the coverage achieved with these test results
- > Added notes to the test results
- > Generated an Aggregate Coverage report for both units
- > Generated a Test Result Management report

Integration Tutorials

Introduction

VectorCAST provides ways to integrate the tools you already use to enforce coding standards, define requirements, and build your applications. The following tutorials demonstrate how to:

- > integrate Gimpel Software's Lint source code analysis engine and perform static analysis on a codebase
- > integrate VectorCAST and VectorCAST/Cover with QA•C® and QA•C++® from Programming Research™ to analyze units and report on coding standards

Integrating Lint

 **Note:** VectorCAST/Lint is in a Maintenance Phase as of August 2020. No more version releases with new features are planned and the product will not be adapted to new operating systems.

VectorCAST is integrated with Gimpel Software's Lint source code analysis engine. Lint is a static source code analyzer that performs module-based or whole-program source code analysis on C/C++ codebases and automatically identifies problems at their source, prior to compiling. The VectorCAST/Lint integration is configured for checking MISRA C (C1), MISRA C (C2), MISRA C (2012), and MISRA C++ standards, and includes an extensive list of embedded compiler option files.

The VectorCAST/Lint integration is available in C/C++ environments and Cover environments.

VectorCAST distributes a Lint executable in the installation, but also allows you to substitute another version of Lint if you are already a Lint user.

Preparing to Run the Tutorial

This Tutorial assumes you are familiar with how to build a C environment. It will take you through the steps but will not provide many details on environment building. You can use any compiler of your choosing; just make sure it is on your PATH before starting VectorCAST, or use the VectorCAST for Visual Studio shortcuts in the Start menu (Windows).

If you are not licensed to build a C environment, then you can run this Tutorial using VectorCAST/Cover. Just add the source files to the Coverage environment and skip the steps where you are instructed to run test cases. You can still perform the Lint steps as instructed in the Tutorial.

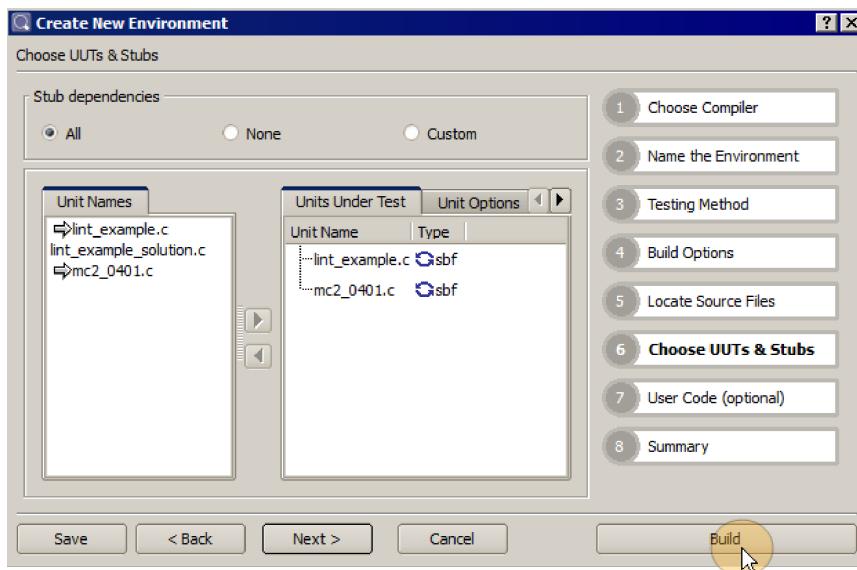
One of the source code units used in this Tutorial was obtained from the Exemplar Suite on the MISRA website. This unit is copyright MIRA Limited, 2006, 2007. We will use this unit to demonstrate VectorCAST/Lint's MISRA capabilities.

You do not need to be a licensed user of Lint to run this tutorial, but you do need a VectorCAST/Lint license (included in Evaluation License Keys).

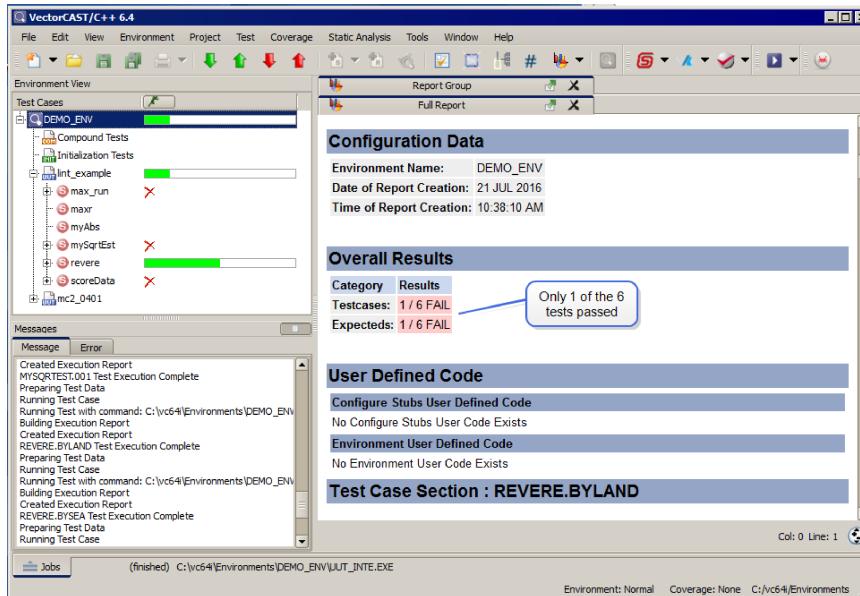
Build the Tutorial Environment

1. Start VectorCAST if it is not already running.
2. From the Welcome page, click **Start Using VectorCAST** and then **Build a VectorCAST/C++ Environment**.

3. On the first page of the Wizard, select your C compiler and click **Next**.
4. Enter your Environment Name and click **Next**.
5. Use Traditional Unit Testing method and click **Next**.
6. Leave the Whitebox option under Build Options checked. Click **Next**.
7. Add the Source directory `$ (VECTORCAST_DIR) \Tutorial\lint` to the Source directories list. Click **Next**.
8. Select the units `lint_example.c` and `mc2_0401.c` as UUTs.



9. Click **Build**.
10. Once the environment builds, select **Test => Scripting => Import Script...** from the Menu Bar and import the test script `lint_example.tst`.
11. Execute all test cases. You see that only 1 out of 6 test cases passed.

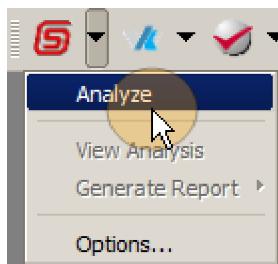


12. Open the test case named **MYSQRTEST.001**, and switch to the Execution Report tab. You see that the test case failed because the actual returned value is 2.1, instead of the expected 2.0. What could be wrong? Examine the other failing test cases.

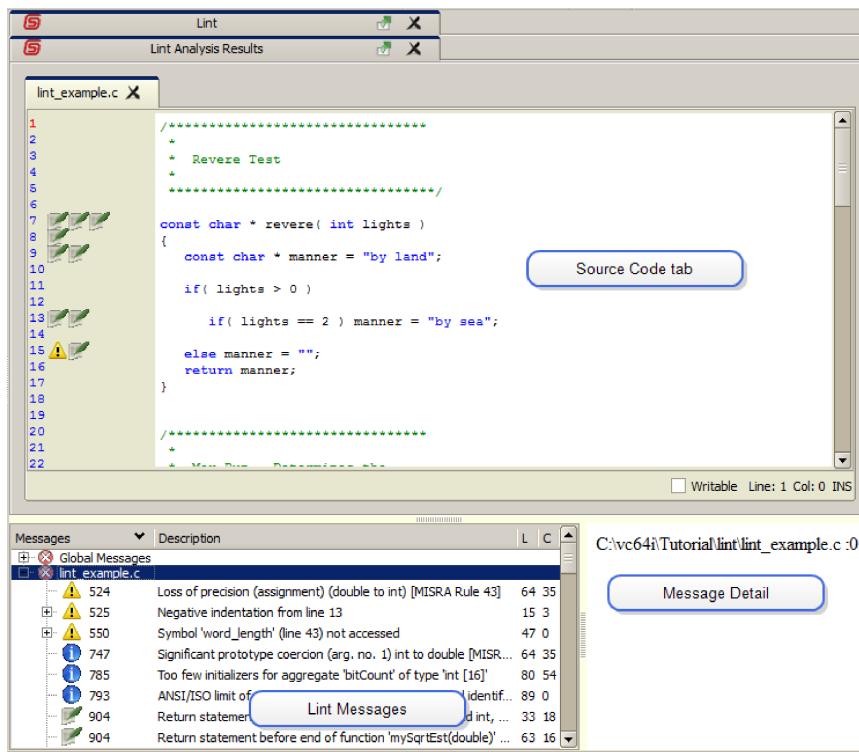
Using Lint to Analyze Source Code

To determine the problems, we are going to run Lint on the unit **lint_example.c** and examine the Lint Analysis Results.

1. In the Test Case Tree, select the UUT **lint_example**. From the Toolbar, go to the Lint Static Analysis icon and select **Analyze** from the drop-down menu.



2. The Lint Analysis Results window opens.

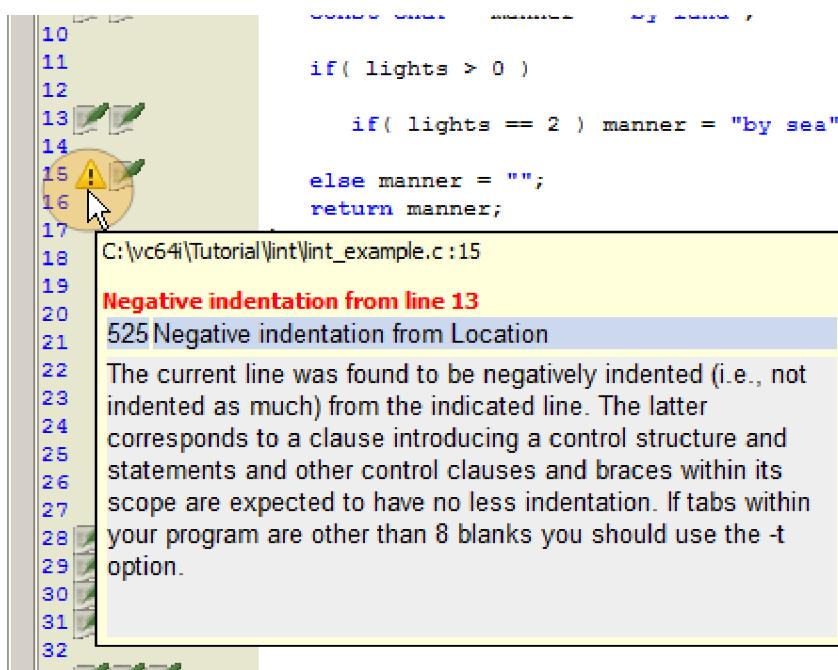


The Lint Analysis Results window has three parts:

- > Source Code tab, which displays `lint_example.c`
- > Lint Messages, which displays the issues found in the unit
- > Message Detail, which shows the path to the unit, and displays details about a particular message, when one is selected.

We are going to correct each Warning issue, and then re-analyze.

3. Hover the mouse over the Warning icon next to line 15. The tooltip explains the problem with this line.



The screenshot shows a code editor window with a yellow warning icon at the top left. A tooltip box is open over line 13, containing the following text:

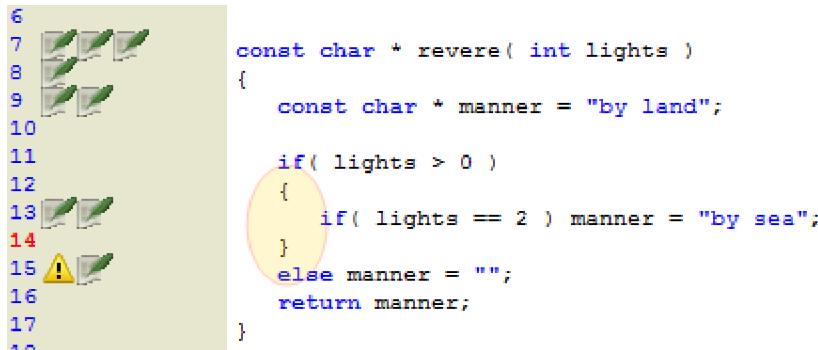
```

C:\vc64i\Tutorial\lint\int_example.c:15
Negative indentation from line 13
525 Negative indentation from Location

The current line was found to be negatively indented (i.e., not
indented as much) from the indicated line. The latter
corresponds to a clause introducing a control structure and
statements and other control clauses and braces within its
scope are expected to have no less indentation. If tabs within
your program are other than 8 blanks you should use the -t
option.

```

- Click the checkbox next to **Writable** Writable Line: 9 Col: 35 INS, and make the correction (add { and } about the if statement) directly in the source file.



The screenshot shows the same code editor window after making the correction. The if statement now has proper braces:

```

6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32

```

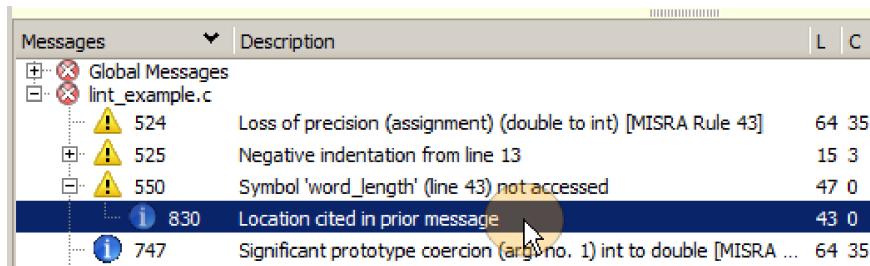
```

const char * reverse( int lights )
{
    const char * manner = "by land";

    if( lights > 0 )
    {
        if( lights == 2 ) manner = "by sea";
    }
    else manner = "";
    return manner;
}

```

- Save the Source file by clicking the **Save** button .
- Expand the message **550 Symbol 'word_length' (line 43) not accessed** in the Lint Messages window, and Click on **830 Location cited in prior message**.



The corresponding line of code is highlighted in the Source Code tab:

```

40  /* max_run(u) returns the maximum run of 0's or 1's in u */
41
42
43
44
45
46
47

```

```

40  /* max_run(u) returns the maximum run of 0's or 1's in u */
41
42
43
44
45
46
47

```

- To correct this problem, change `w` to `word_length` in the return statement.

```

40  /* max_run(u) returns the maximum run of 0's or 1's in u */
41
42
43
44
45
46
47

```

```

40  /* max_run(u) returns the maximum run of 0's or 1's in u */
41
42
43
44
45
46
47

```

- Save** the source file ().
- Go to the Lint Static Analysis icon on the Toolbar and select **Analyze** from the drop-down menu again to re-analyze. A Warning remains on line (64).
- Fix the issue in a similar way:
On line 64, there is a loss of precision in the type being used for **error**.

The screenshot shows the VectorCAST IDE interface. In the top window, titled 'lint_example.c', there is a line of code with a yellow warning icon. A callout bubble points to the line number 64 with the text 'Warning on line 64'. In the bottom window, titled 'Messages', there are two entries: one for warning 524 (Loss of precision) at line 64, column 35, and another for note 747 (Significant prototype coercion) at line 64, column 35.

```

59
60
61
62
63
64
65
66
67
68
69
    double x = 1.0; /* initial estimate */
    int error;
    if( y <= 0 ) return 0; /* ensure positive value */
    while( myAbs( error = y - x * x ) > .0005 )
        x = x + error / (2*x); /* adjust estimate */
    return x;
}

```

Change the type for **error** from int to double:

The screenshot shows the VectorCAST IDE interface with the same source code as before, but now the variable 'error' is declared as a 'double' type. The IDE highlights the change with a green circle and a cursor icon.

```

59
60
61
62
63
64
65
66
67
68
69
{
    double x = 1.0; /* initial estimate */
    double error;
    if( y <= 0 ) return 0; /* ensure positive value */
    while( myAbs( error = y - x * x ) > .0005 )
        x = x + error / (2*x); /* adjust estimate */
    return x;
}

```

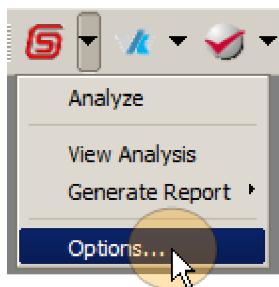
That change corrects the issue.

11. Save the source file and re-analyze. No Warnings found!
12. Before you execute the test cases again, you need to rebuild the environment because we have changed the source code. Choose **Environment => Rebuild Environment**.
13. Execute the test cases, and all pass.

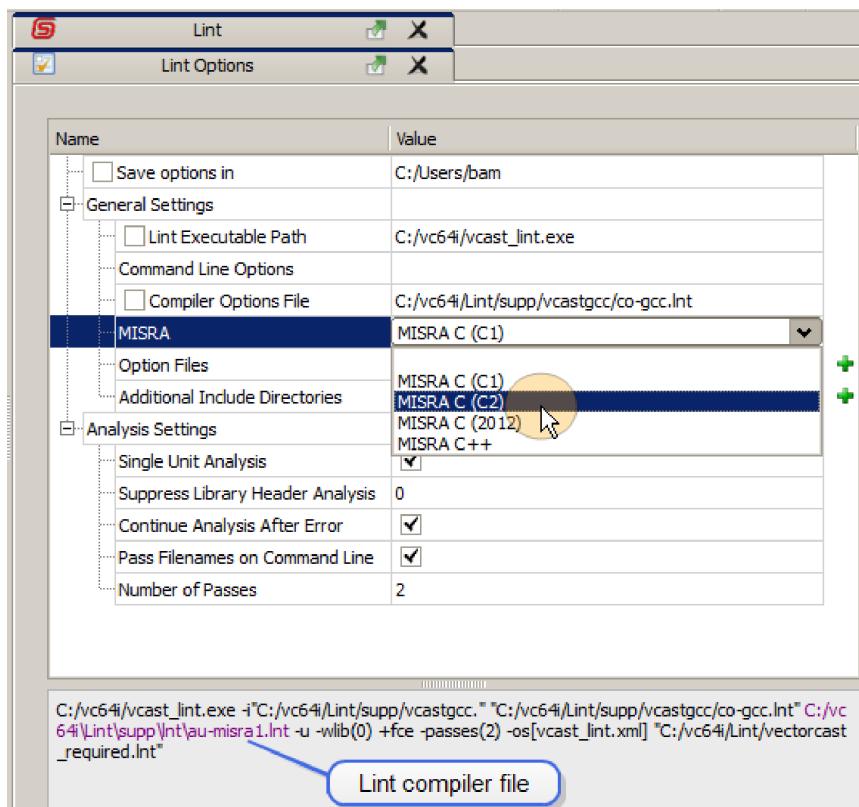
Using VectorCAST/Lint's MISRA Compliance Checking

Another unit has been included in the environment which will help us demonstrate VectorCAST's integrated MISRA compliance checker. This unit was obtained from the MISRA Exemplar Suite, available on the MISRA website's Online Bulletin Board. The source file is named mc2_0401.c, which refers to MISRA (C2) Rule 4.1, "Only those escape sequences that are defined in the ISO C standard shall be used."

1. From the Toolbar, go to the Lint Static Analysis icon and select **Options..** from the drop-down menu. The Lint Options tab opens.

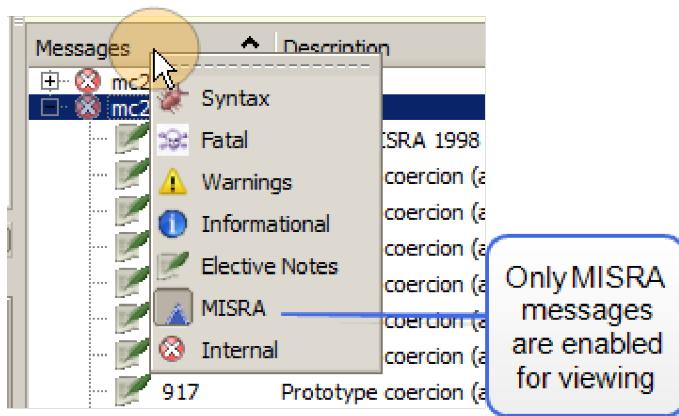


- Click in the Value column next to the option MISRA, and choose **MISRA C (C2)** from the drop down.

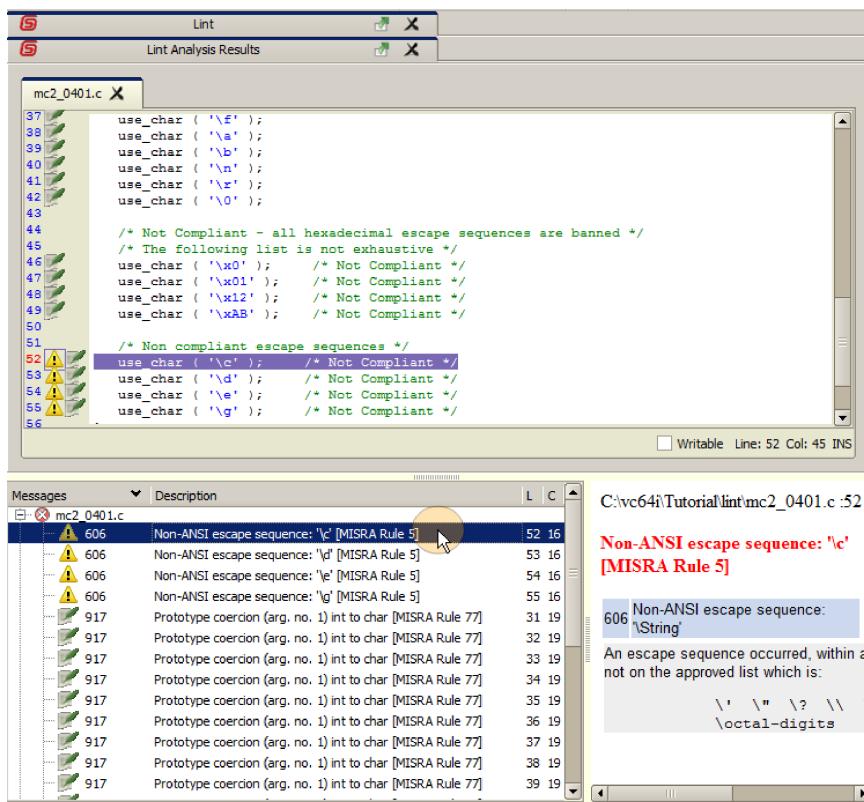


The colored text at the bottom reflects the Lint compiler file constructed for MISRA C (C2) that is added to the command line to call Lint.

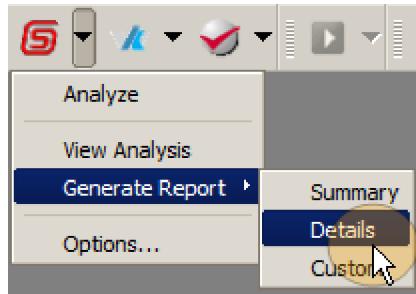
- Select the unit **mc2_0401** in the Test Case Tree.
- From the Toolbar, go to the Lint Static Analysis icon and select **Analyze** from the drop-down menu.
The change to the Lint Options is saved automatically before analysis.
- In the Lint Analysis Results window, a tab for the unit **mc2_0401** opens. The Lint Messages window shows many messages.
- To filter the Lint Messages window so that only MISRA messages are displayed, right-click the column heading **Messages** and turn off each type, one by one, except MISRA.



7. Now that messages of the other types are turned off, there is only one Warning message per line of non-compliant code. The picture below shows the Lint Analysis Results window with the first occurrence of message 606 selected.



8. From the Toolbar, go to the Lint Static Analysis icon and select **Generate Report => Details** from the drop-down menu.



The Detailed Lint Summary Report from the most recent analysis is displayed.

A screenshot of the VectorCAST/Lint Detailed Report window. The title bar says 'Lint Reports'. The main area is titled 'VectorCAST/Lint Summary Report' and contains a table with the following data:

File Name	Total Issues	Syntax	Fatal	Warnings	Informational	Elective Notes	MISRA	Internal
mc2_0401.c	27	-	-	-	2	-	25	-
mc2_types.h	1	-	-	-	-	-	1	-
Global Messages	1	-	-	1	-	-	-	-
Totals	29	-	-	1	2	-	26	-

The detailed report table below lists specific MISRA violations for 'mc2_0401.c':

Name	Summary	Syntax	Fatal	Warnings	Informational	Elective Notes	MISRA	Internal
mc2_0401.c	27	-	-	-	2	-	25	-
960(M) (L:0C0)	Violates MISRA 1998 Required Rule 90, Disallowed definition for macro '__asm__'							
917(M) (L:31C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:32C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:33C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:34C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:35C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:36C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:37C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:38C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:39C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:40C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:41C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:42C19)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:46C20)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:47C21)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							
917(M) (L:48C21)	Prototype coercion (arg. no. 1) int to char [MISRA Rule 77]							

Homework

Your homework is to figure out how to make a custom Lint Analysis report that has only MISRA messages included in it.

Tutorial Summary

In this tutorial, you:

- > Analyzed a unit using the Lint analyzer integrated with VectorCAST
- > Corrected logic errors, incorrect type being used, and incorrect array initialization found by Lint in the source code, which were causing test cases to fail

- > Achieved all tests passing after correcting errors and rebuilding the environment
- > Analyzed a source file using the built-in MISRA C (C2) capabilities
- > Created a Lint Analysis Report

Integrating PRQA

VectorCAST is integrated with Programming Research's market leading C and C++ source code analysis technologies. The QA•C and QA•C++ tools are consistently recognized worldwide as the most powerful, most robust, and most technically advanced solutions available today for analyzing source code and enforcing pre-defined coding standards (such as MISRA and JSF++). If you are already a licensed user of QA•C or QA•C++, then you can run this tutorial to discover how easy it is to reach the PRQA functionality when using VectorCAST or VectorCAST/Cover.

This tutorial takes you through the steps to configure PRQA for use with VectorCAST. It is assumed that you have run one of the C or C++ tutorials in this guide. The tutorial refers to a C source code unit, but if you choose to use QA•C++, you can easily make the substitutions.

Preparing to Run the Tutorial

If you have not run any of the C or C++ tutorials, you will need to do so first. If desired, you do not need to go any further than building the environment; test cases are not needed to use the PRQA tools with VectorCAST.

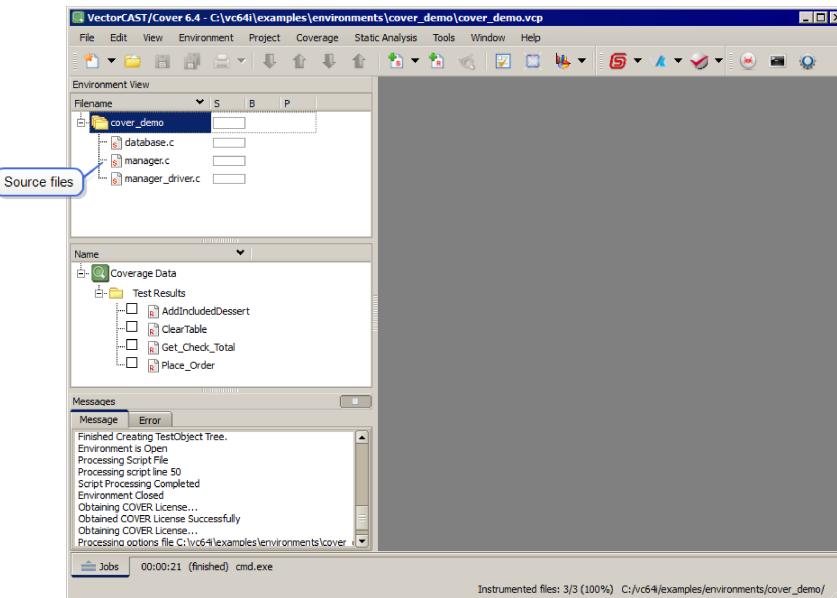
In addition, you must be a licensed user of QA•C or QA•C++ to perform this tutorial.

Using QA•C++ to Analyze Source Code

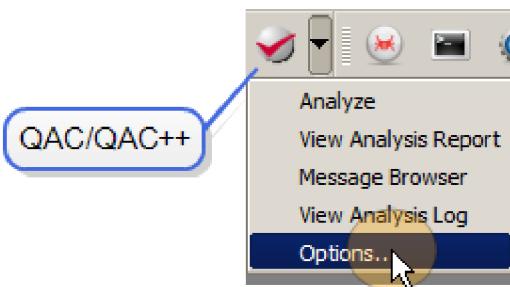
This tutorial will use the cover_demo environment that you created in the VectorCAST/Cover Tutorial. You can substitute any other C or C++ environment or Cover environment.

1. Start VectorCAST if it is not already running.
2. Select **File => Recent Environments**, and choose **\$(VECTORCAST_DIR)\examples\environments\cover_demo\cover_demo.vcp**.

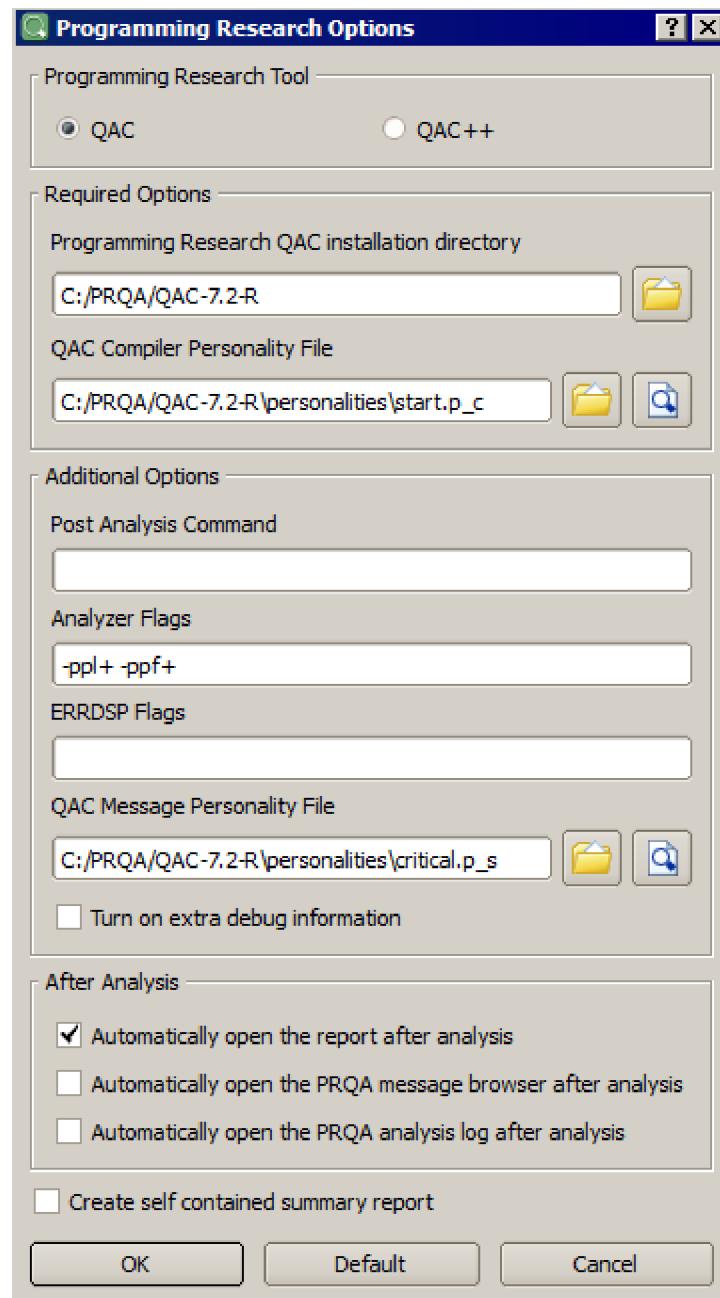
This coverage environment has three source files: `database.c`, `manager.c` and `manager_driver.c`. For the purposes of this tutorial, it doesn't matter if you have any test results selected.



3. Before analyzing any units, you should ensure that the PRQA options are set correctly. From the Toolbar, go to the QAC/QAC++ icon and select **Options...** from the drop-down menu.



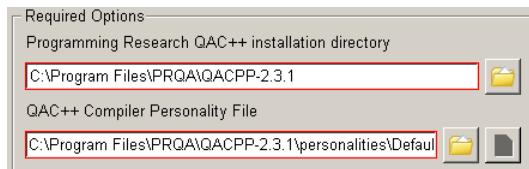
The Programming Research Options dialog opens.



VectorCAST detects whether your environment is C or C++, and automatically selects the language mode for the QAC tool.

If you are using a different version or if you installed it in a location other than the default, then the

paths are outlined in red. Just click the **Browse** button  and navigate to the correct location.



4. Click **OK** to close the Options dialog.
5. To start the analysis on the UUTs, multi-select both UUTs (**manager** and **database**), or just select the top level, **TUTORIAL**.
6. Click the **Analyze** button  on the Integrated Tools toolbar.

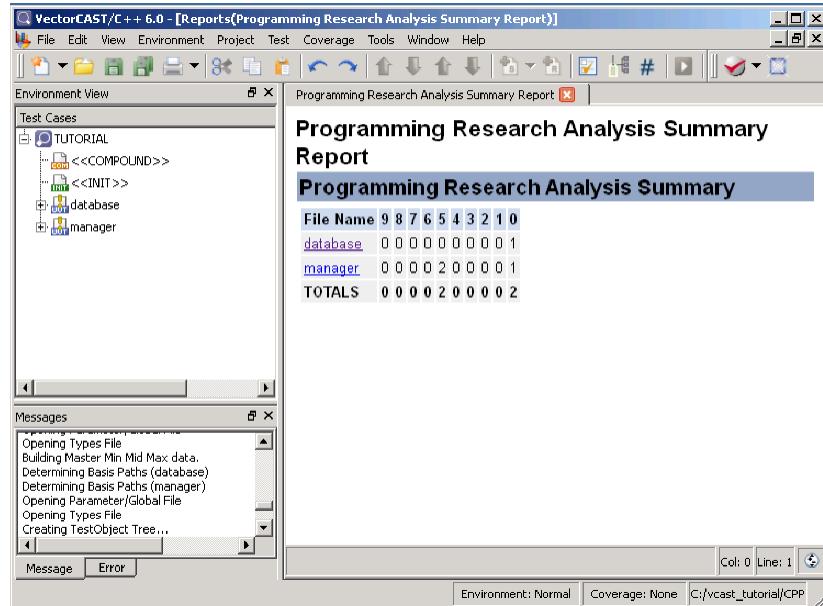


If you see the FLEXIm License Finder dialog, then PRQA cannot find a license.



Cancel out of the dialog, and exit VectorCAST. Then set up the **LM_LICENSE_FILE** environment variable for both VectorCAST and the Programming Research Tools.

The Analyze button animates, , to indicate that QA•C++ is analyzing in the background. You can use VectorCAST for other tasks while the analysis is taking place. When the analysis is finished, the Summary report is displayed in the Report Viewer:



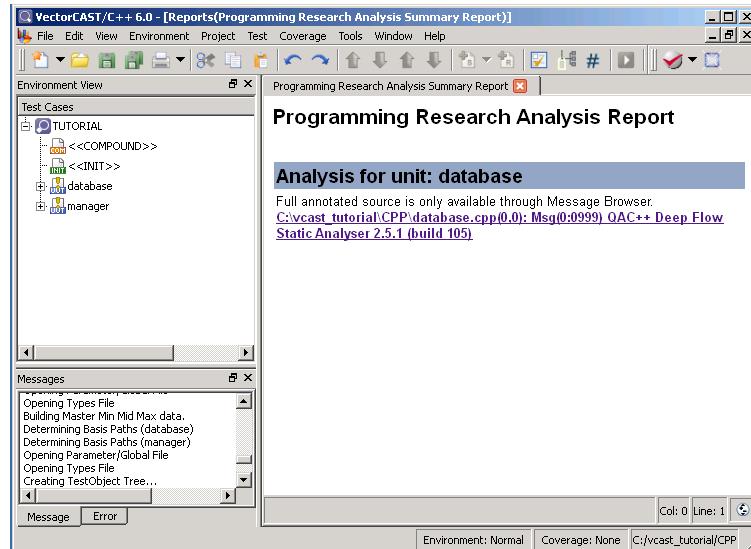
The Summary report lists each unit analyzed, and provides a link. The numbers to the right are the various error levels, with 9 being the most serious. Our results show that 1 occurrence of a level 0 error exists for each unit (because these are *very simple* source files). You will follow the trail to determine the meaning of the “error”.

7. Click the link **database** in the Summary report.

The Analysis report for the unit **database** opens in the Report Viewer. You have drilled down one level.

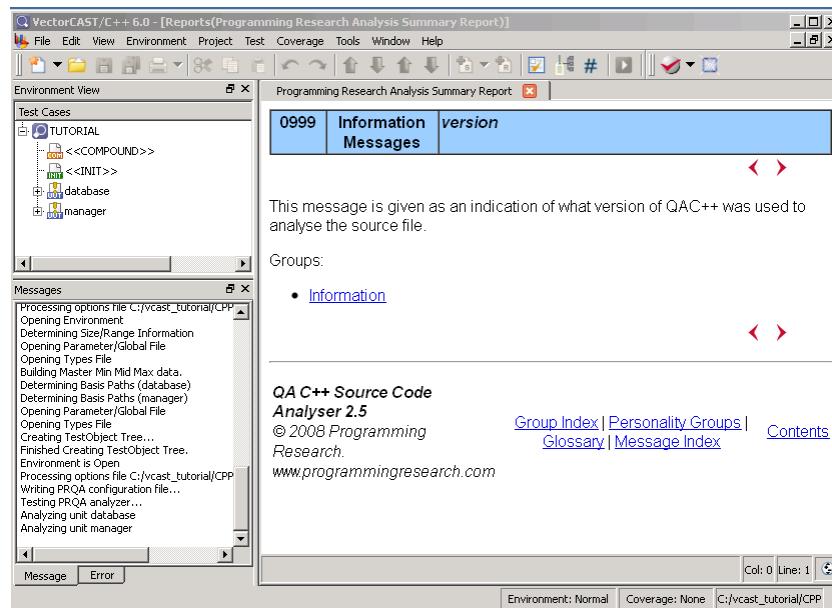


Note: If you ran the wrap_compile tutorial, you will see the instrumented version of the source file in the report, because the **Instrument in place** option was turned on.



The “error” in the source file is message 0:9999.

- Click the link **C:\vcast_tutorial\CPP\database.cpp(0):Msg(0:0999) QAC++ Deep Flow Static Analyser 2.5.1 (build 105)**. Information about that message appears in the Report Viewer. You have drilled down another level.

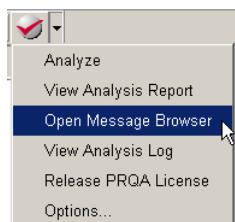


- To drill down one more level, click the link **Information**.

The “error” is merely a message with information about the version of QA•C++ used for analysis.

Message	Message Text
0999	version

- To get back, right-click the report and choose **Back** or click the Undo button .
- Click the little arrow  to the right of the Analyze button, and choose **Open Message Browser** from the list:



The QA•C++ Message Browser opens.

- Expand Information Messages, and click on **999: 1%**.

The source code for `database.cpp` opens in the Source tab, as you would expect.

File	Line	Col	Msg	Message Text	#
database.cpp	0	0	999	QAC++ Deep Flow Static Analyser 2.5.1 (build 105)	1
manager.cpp	0	0	999	QAC++ Deep Flow Static Analyser 2.5.1 (build 105)	1

13. Exit the Message Browser.

Tutorial Summary

In this tutorial, you:

- > Configured Programming Research's QA•C++ to run with VectorCAST
- > Analyzed both UUTs in an environment you created earlier
- > Drilled down from the Summary report to the detailed information about the error message
- > Opened the PRQA Message Browser from VectorCAST

Enterprise Testing Tutorial

Introduction

Enterprise Testing provides a way to manage testing over an entire project or organization. Many projects have no ability to modify source code with confidence that it will still work, due to a lack of information on which tests to re-run. VectorCAST Enterprise Testing identifies which tests need to be re-run based on the source code changes you have made, and ensures that you have not broken existing functionality while fixing a bug or adding new functionality. Enterprise Testing makes the testing process easy to run and gets the results sooner.

Enterprise Testing:

- > provides a single point-of-control for all unit- and integration-test activities
- > enables you to create regression test suites by importing previously-developed test environments from VectorCAST/C++ and VectorCAST/Ada
- > provides easy-to-review logs and summary reports with color-coded pass/fail criteria that highlight the status of each test within the regression suite
- > supports testing of multiple source code baselines and releases
- > provides easy identification of testing trends and regressions
- > provides a full-featured command line interface
- > enables you to open an individual environment in order to diagnose and fix test failures
- > is integrated with source code management (SCM) tools to allow a single point of control for all test artifacts

How Enterprise Testing Works

Enterprise Testing uses the same user interface as the other products in the VectorCAST family: the familiar tree on the left displays the project hierarchy, the MDI window displays configuration, reports or environments that are open, and the Message window displays the test build and execution output.

Enterprise Testing takes existing VectorCAST environments and imports them into a VectorCAST *project*. Individual environments can be grouped into Environment Groups and Test Suites to reflect your project architecture. An environment can be a member of multiple groups, and an Environment Group can be assigned to multiple Test Suites. Because Environment Groups and Test Suites are easily duplicated, the same tests can be run using various source code baselines, on different host platforms, or with a different compiler or embedded target.

As test cases are executed in the environments, data is stored which enables you to see historical trends. Each environment's build status and duration and test case execution status and duration, as well as the code coverage achieved is stored.

Terminology

The **VectorCAST project** is the repository for all information created with Enterprise Testing. It is analogous to the environment in VectorCAST/C++ or VectorCAST/Ada. A project is generally organized by the user to reflect the hierarchy of the application.

An **environment group** is a logical collection of VectorCAST environments.

A **test suite** is a logical collection of environment groups.

A **compiler node** is where all compiler-specific VectorCAST options are stored. The compiler node knows the default options for a particular compiler, and enables you to change settings or override default compiler options for your project needs.

A **data source** is a location where all the work is done, that is, where the VectorCAST environments are built and executed. It may be located in the project directory or in some other location, even on a remote machine. A project's data sources can be local to one machine or distributed across multiple machines. Data from remote data sources can be pushed back to the central repository. The status from all data sources in a project is combined into the Management Summary Report.

Now that you have some idea about what Enterprise Testing can do for you, it's time to try it out. The following tutorial demonstrates how to:

Creating a VectorCAST Project

This tutorial takes you through all the steps necessary to create a VectorCAST project.



Note: This tutorial uses a VectorCAST project with imported C environments. However, there are only minor differences when using Ada environments. You should build environments using the most convenient C or Ada compiler on your system; source code and scripts are provided for both.

What You Will Accomplish

In this tutorial, you will:

Preparing to Run the Tutorial

Before you can run this tutorial, you need to:

- > Create a local directory named `vcast_tutorial`.
- > Set up the two source code baselines using the Tutorial files. In baseline1 are the standard tutorial files. In baseline2, manager.c (manager.adb) has been modified so that the entrée STEAK costs \$114.00 instead of \$14.00. *This change will be detected as a bug in the source code after executing the unit test environment in Enterprise Testing.*
- > Build two C or Ada environments using the tutorial files and a compiler on the local machine.

Directions for these preparations are given below.

Setting Up the Source Files

Before you can run this tutorial, you need to decide on a working directory and copy into this directory the Tutorial files from the VectorCAST installation directory. For efficiency, this tutorial assumes a working directory named `vcast_tutorial` located at the top level in your file tree. You can, however, locate this directory any place you want, and make the necessary translations in the text as you progress through the tutorial.

The first step in this tutorial involves quickly creating two unit test environments, using either a C or Ada

compiler. A zip file named `enterprise_unit_testing.zip` is provided in the VectorCAST installation directory, `tutorial/enterprise_unit_testing` sub-directory. It contains the familiar C and Ada tutorial files, as well as a modification of those files in another baseline. A directory with environment scripts (.env) and test scripts (.tst) is provided for your convenience.

In Windows Explorer

Using Windows Explorer, navigate to `%VECTORCAST_DIR%\Tutorial\enterprise_unit_testing\enterprise_unit_testing.zip` and copy and paste this zip file to `vcast_tutorial`. Unzip (extract) it there.

In a UNIX Shell

If necessary, change directory to `vcast_tutorial`. Then enter:

```
$ cp $VECTORCAST_DIR/Tutorial/enterprise_unit_testing/enterprise_unit_testing.zip.
$ unzip enterprise_unit_testing.zip
Archive: enterprise_unit_testing.zip
  creating: ENTERPRISE_UNIT_TESTING/baseline1/
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/ctypes.h
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/database.adb
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/database.ads
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/database.c
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/manager.adb
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/manager.ads
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/manager.c
  inflating: ENTERPRISE_UNIT_TESTING/baseline1/types.ads
  creating: ENTERPRISE_UNIT_TESTING/baseline2/
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/ctypes.h
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/database.adb
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/database.ads
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/database.c
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/manager.adb
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/manager.ads
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/manager.c
  inflating: ENTERPRISE_UNIT_TESTING/baseline2/types.ads
  creating: ENTERPRISE_UNIT_TESTING/regression_scripts/
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/ADA_DATABASE.env
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/ADA_DATABASE.tst
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/ADA_MANAGER.env
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/ADA_MANAGER.tst
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/C_DATABASE.env
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/C_DATABASE.tst
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/C_MANAGER.env
  inflating: ENTERPRISE_UNIT_TESTING/regression_scripts/C_MANAGER.tst
  creating: ENTERPRISE_UNIT_TESTING/unit_test_envs/
```

In the `vcast_tutorial` directory, the unzipped file creates a directory named `ENTERPRISE_UNIT_`

TESTING. Below that are the two source code directories (`baseline1` and `baseline2`) which contain both C and Ada source files, a directory named `regression_scripts` which contains environment scripts (.env) and test scripts (.tst), and an empty directory named `unit_test_envs`, in which you will build two unit test environments.

Name	Date modified	Type	Size
baseline1	2/26/2018 3:45 PM	File folder	
baseline2	2/26/2018 3:45 PM	File folder	
regression_scripts	2/26/2018 3:45 PM	File folder	
unit_test_envs	10/4/2010 4:29 PM	File folder	

Compile Source Code (Ada only)

If you plan to create two Ada unit test environments, you will need to compile the source code first. You need to compile both the sources in `baseline1` and `baseline2`.

Build the Environments

This tutorial assumes you know how to build a VectorCAST/C++ or VectorCAST/Ada environment. If you have never used VectorCAST before, you should come back to this tutorial later and instead follow one of the Basic Tutorials earlier in this *Interactive Tutorials* guide.

1. With your compiler on your PATH, start VectorCAST/C++ (or VectorCAST/Ada).
2. Choose **File => Set Working Directory** and navigate to `vcast_tutorial/enterprise_unit_testing\ENTERPRISE_UNIT_TESTING\unit_test_envs`.
This directory is where you will build two unit test environments, either C or Ada.
3. Bring up a Create New Environment Wizard (**File => New => C/C++ Unit Test Environment** (or **Ada Host Unit Test Environment**)).
4. Set compiler if necessary.
5. Load environment script `../regression_scripts/C_MANAGER.env` (or `ADA_MANAGER.env`) into the Wizard.
6. Build.
7. Import test script `../regression_scripts/C_MANAGER.tst` (or `ADA_MANAGER.tst`).
8. Execute test case and verify it passes.
9. Close environment.
10. Repeat steps 3-9, this time building ENV_DATABASE and importing the test script for that environment.

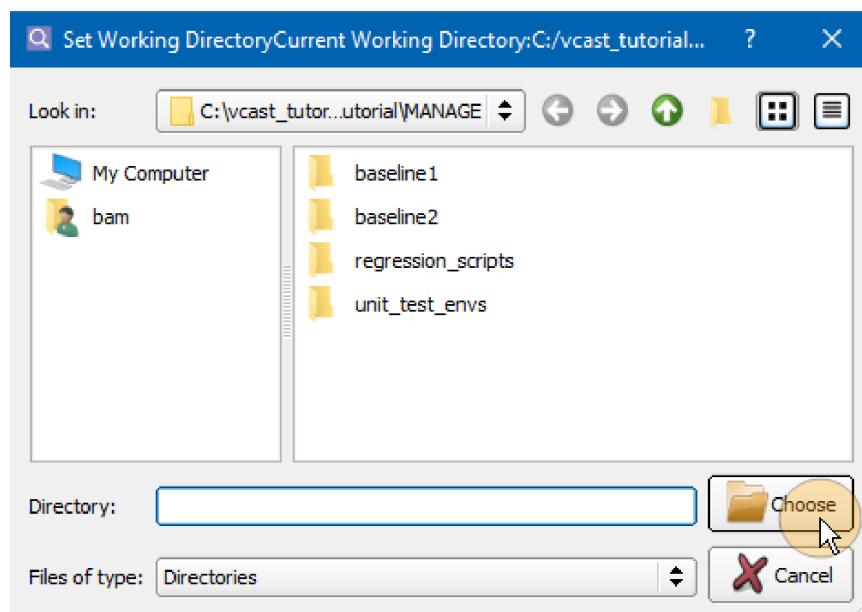
You are now ready to create a VectorCAST project and import the two environments.

Using the VectorCAST Project Wizard

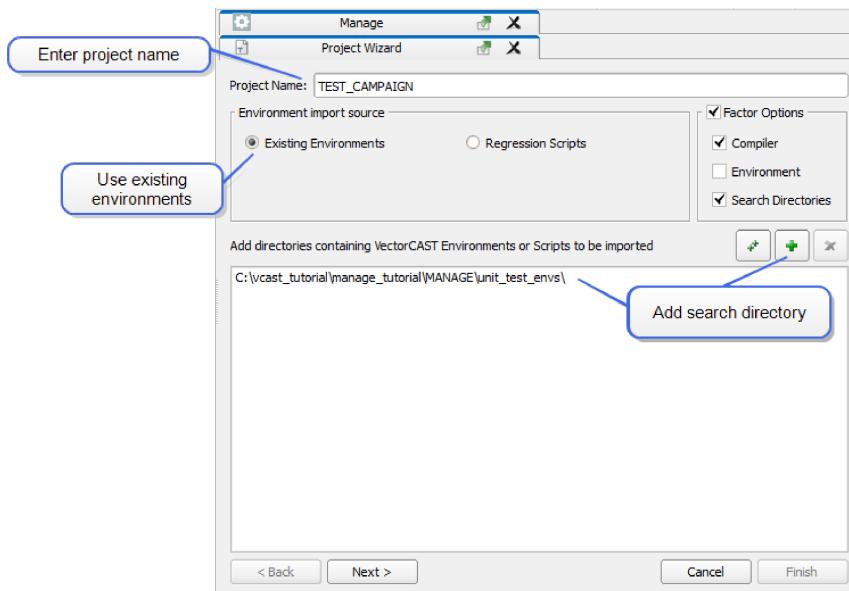
In this tutorial, you will use the VectorCAST Project Wizard to create a simple VectorCAST project. Creating a VectorCAST project involves:

- > setting the working directory
- > deciding between importing existing, built environments or importing using regression scripts
- > deciding which environment configuration options to filter up to a higher, more abstract level
- > specifying a data source
- > verifying the group names and test suite names are satisfactory
- > finishing

1. Select **File => Set Working Directory** and navigate to **vcast_tutorial/manage_tutorial/MANAGE**.



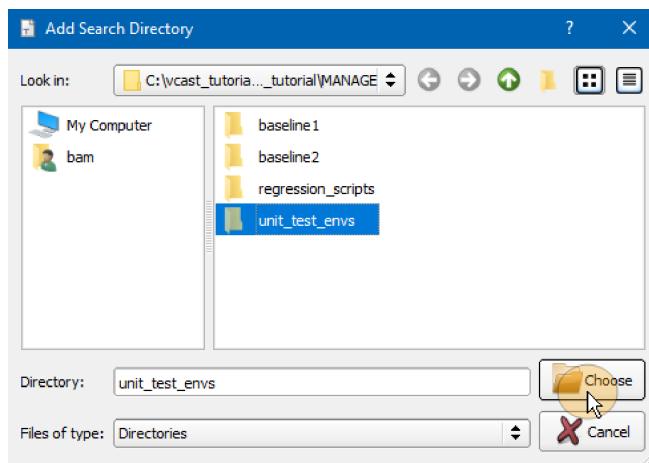
2. Click **Choose**.
3. Select **File => New => VectorCAST Project =>From Existing Environments**. The Project Wizard opens.
4. Give the project the name **TEST_CAMPAIGN**.



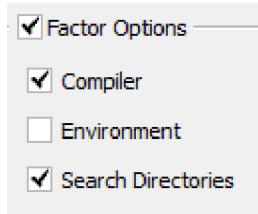
- Leave the option **Environment import source** set to **Existing Environments**.

This option specifies whether VectorCAST should look for built environments to import or to look for sets of regression scripts – the shell/batch file (.sh/.bat), the test script (.tst), and the environment script (.env). In our case, we have two existing unit test environments, so we leave the option on **Existing Environments**.

- Click the **Add Search Directory** button and select the directory where the existing environments reside, **unit_test_envs**.



- Click **Choose**.
- Ensure **Factor Options** is checked, and add a checkmark next to **Search Directories**, so that both Compiler options and Search Directories are checked.



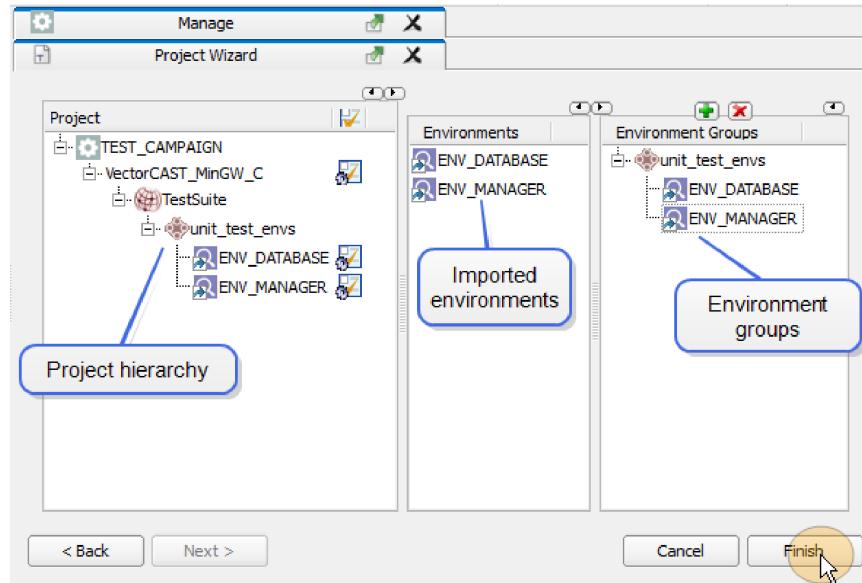
Factoring options makes it easy to reuse environments in multiple contexts in the VectorCAST project. Options set at the environment level are “bubbled up” to higher levels in the project hierarchy, where they can be easily modified to affect many environments at once.

There are three types of factoring:

- > An environment's *Compiler options* are factored to the compiler node. Default settings are discarded because the Compiler node already has those. Overridden settings are preserved. Compiler option differences between two environments result in multiple compiler nodes.
- > An environment's *Environment options*, such as Whitebox, coverage type, Build, Execute, Coverage, and Report options are factored up to the Test Suite node. Environment options include all options except compiler options and Source directories. Environment option differences between two environments result in multiple Test Suite nodes.
- > An environment's *Search directories* and Type-handled directories are factored to the Test Suite level. (Library Include directories are factored to the compiler level.) Factoring Search Directories makes it easy to change the source code baseline used to build the unit test environments, which is exactly what we want to do in this tutorial.
- > With no factoring, environments are just put in groups according to their directory of origin. All options remain at the individual environment level.

9. Click **Next** to move to the next page of the Wizard.
10. VectorCAST begins importing the unit test environments that it finds in the specified Search directories.

When finished importing, the Wizard looks like this:

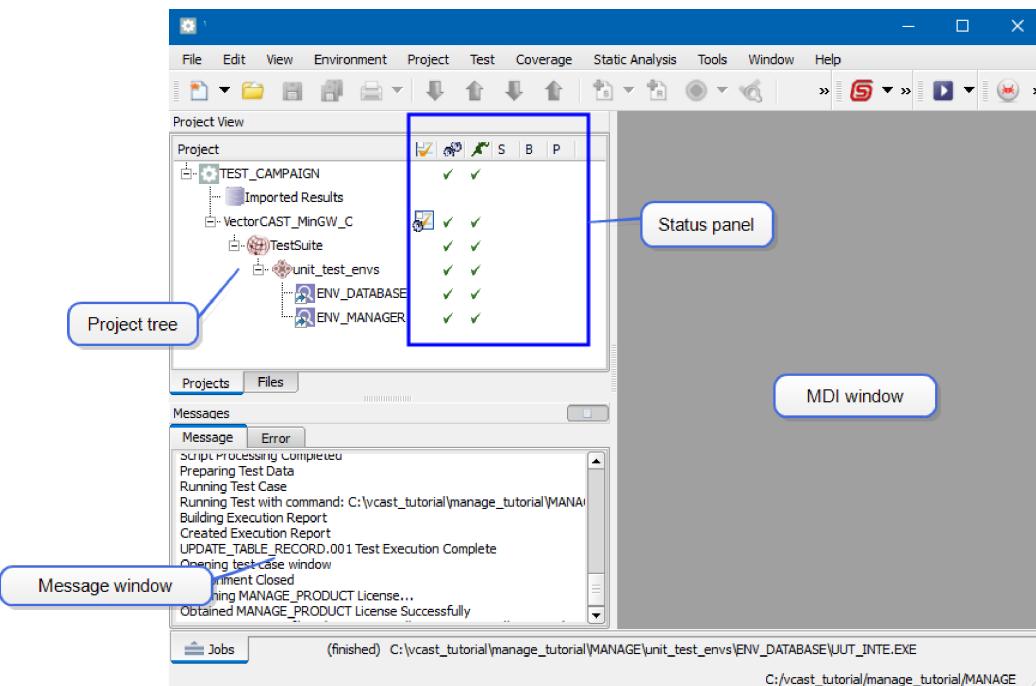


If you created Ada environments, the environment icons are orange instead of blue.

This page of the Wizard can be used to rename nodes in the hierarchy or reorganize the hierarchy by dragging and dropping.

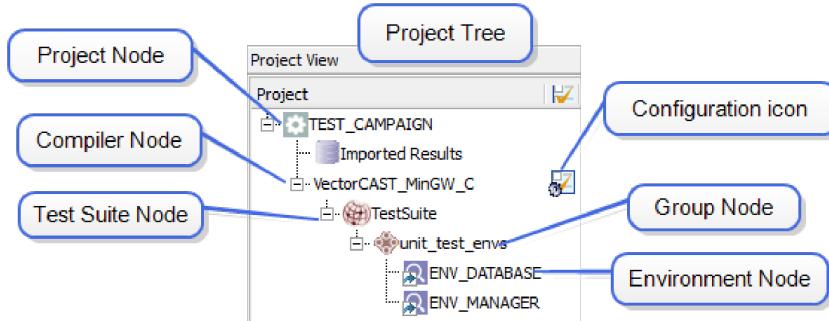
11. Click **Finish**.

The project opens in the main window.



The Project Tree

The Project Tree shows the project architecture and enables you to apply an action to a level and those below it. Each level in the hierarchy has a context-sensitive right-click menu.



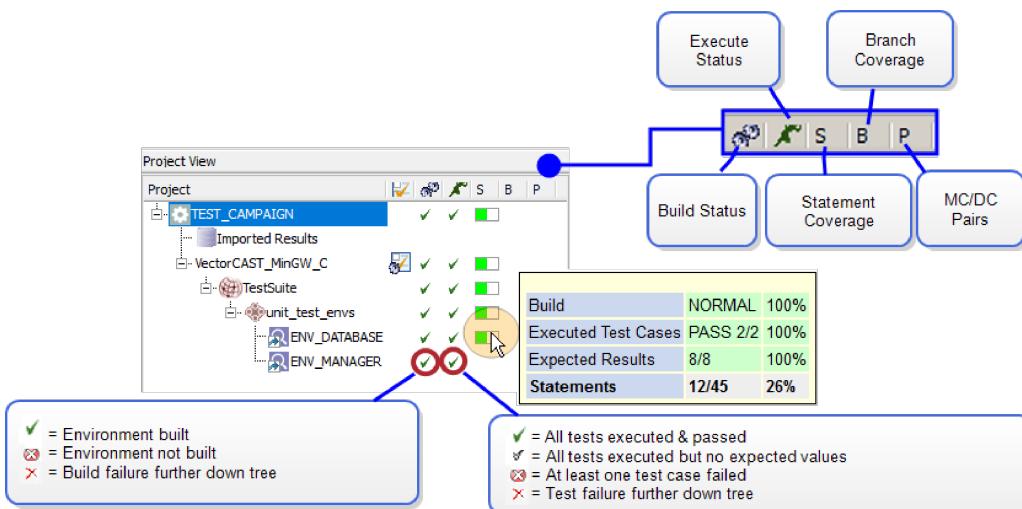
Node	Description
<code>TEST_CAMPAIGN</code>	Project node. Root node of the project.
<code>VectorCAST_MinGW_C</code>	Compiler node. Each compiler node is associated with one compiler template.
<code>TestSuite</code>	Test Suite node. The Wizard creates one Test Suite node for every set of common options, naming them “Configuration_01,” “Configuration_02,” etc.
<code>unit_test_envs</code>	Environment Group node.
<code>ENV_MANAGER</code>	Environment node. The Environment node utilizes several icons to identify an environment.

Node	Description
	<ul style="list-style-type: none"> C/C++ environment Ada environment Cover environment TUTORIAL_C Disabled environment Source file has changed Unbuilt environment Locked environment Monitored environment Invalid environment
	Configuration icon. Any node that can have a custom configuration has this icon. Hover mouse over to see configuration for the node.

The Status Panel

The Status Panel is directly to the right of the Project tree, and displays the testing status at the current time. The status of the upper levels in the hierarchy depends on the status of the lower levels. If something below a level is failing, then that whole branch of the tree is failing as well. For the root node to be passing, all levels below must be passing.

The status panel is divided into columns, each of which shows the status of a different operation: Environment build, test execution, Statement coverage, Branch coverage, and MC/DC pairs status.



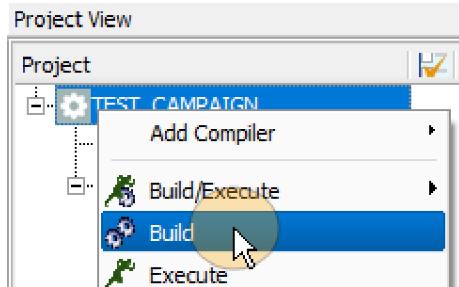
When MC/DC coverage type is applied to an environment, the MC/DC branches are displayed in the

Branch column and the pairs are displayed in the MC/DC Pairs column. DO-178B Level A coverage type uses the Statement column, the Branch column for the MC/DC branches, and the MC/DC pairs column. Similarly, DO-178B Level B coverage type uses the Statement and Branch columns.

Building and Executing the Environments

Now that you have built the VectorCAST project, you are ready to build the environments.

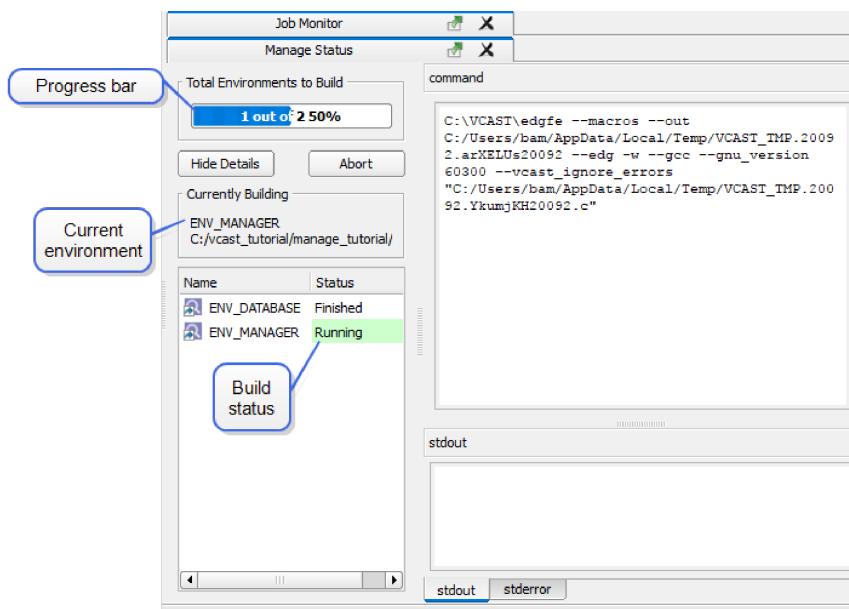
1. Right-click the root node of the project (TEST_CAMPAIGN) and select **Build** from the context menu.



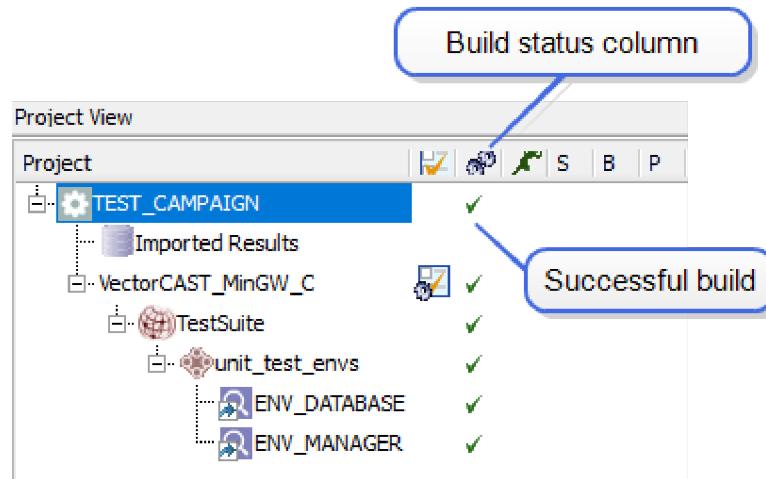
This action causes VectorCAST to build the two environments and import their associated test script.

2. The Manage Status viewer opens in the MDI window. While an environment is building or executing, VectorCAST puts an environment lock (🔒) on the environment to prevent any other processes from accessing the environment at that time.

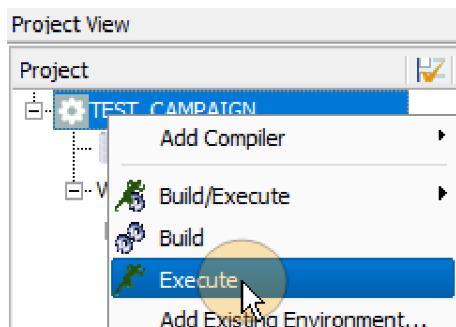
The Manage Status viewer provides detailed real-time environment build and execution information, showing the progress, outcome and output for environments in a VectorCAST project.



When finished building, you see the Build Status column has green checkmarks ✓ all the way up.

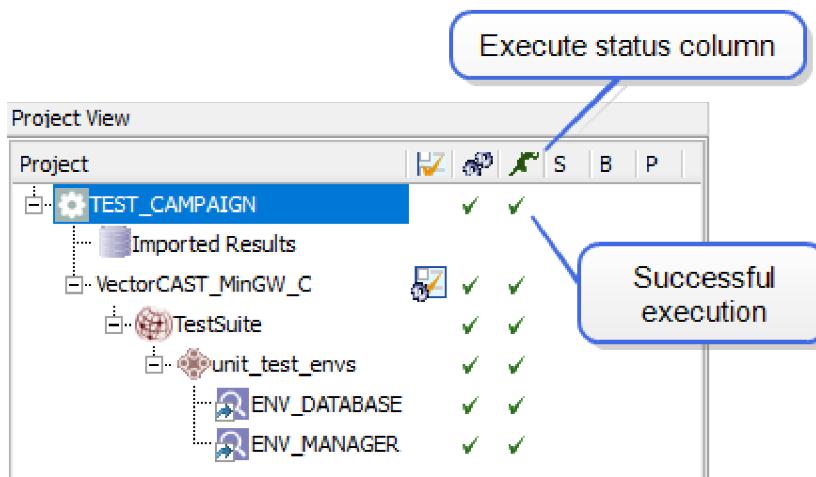


- Right-click the root node of the project (TEST_CAMPAIGN) and select **Execute** from the context menu.



This action causes VectorCAST to execute batch all environments. The Manage Status viewer again opens in the MDI window, providing detailed real-time execution information.

When finished executing, you see the Execute Status column has green checkmarks all the way up.



- Right-click on the root node and choose **Reporting**. The Manage Report opens in the MDI window.

Manage Report

Configuration Data

Date of Report Creation: 14 FEB 2018
Time of Report Creation: 4:46:23 PM
Version: 18 (02/04/18) VectorCAST version number

Status Section

	BUILD	BUILD TIME	EXPECTED	TESTCASES	EXECUTE TIME
ALL	2/2 (100%)	00:22	-	-	00:03
VectorCAST_MinGW_C	2/2 (100%)	00:22	-	-	00:03
TestSuite	2/2 (100%)	00:22	-	-	00:03
unit_test_envs	2/2 (100%)	00:22	-	-	00:03
ENV_DATABASE	NORMAL	00:11	-	-	00:01
ENV_MANAGER	NORMAL	00:11	-	-	00:02

Table rows correspond to Project Tree

The Manage Report provides a summary of build, execution, and coverage results in table form. The report contains two sections:

- > The Configuration Data section providing the time and date the report was generated and the VectorCAST version used, and
- > The Status section providing build, execution and coverage metrics. The report includes the parents of the level generating the report.

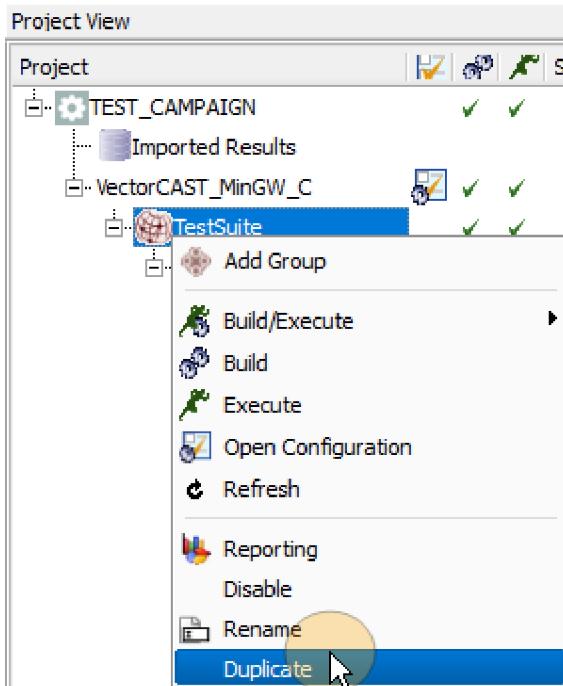
Enterprise Testing makes it easy to execute the same environment with a different Source baseline, a different compiler, or a different Test Suite configuration. And that is exactly what we are going to do in the next section.

Modifying the Project

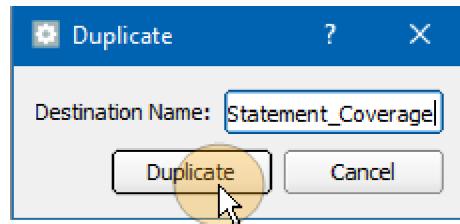
In this section we will duplicate the Test Suite to add code coverage. We could just put code coverage on the existing Test Suite and rebuild the environments, but doing it this way shows you how easy it is to run the same environments in a different configuration.

Modifying a VectorCAST project makes a change to the project file (**TEST_CAMPAIGN.vcm**). When you open a project, you have the permissions to modify the project file. If another person opens the project, that person has permissions to modify the project. Whoever tries first will need to disconnect the other user in order to proceed with the modification. (We don't want two users changing the project file at the same time!)

1. Right-click the Test Suite node and select **Duplicate** from the context menu.

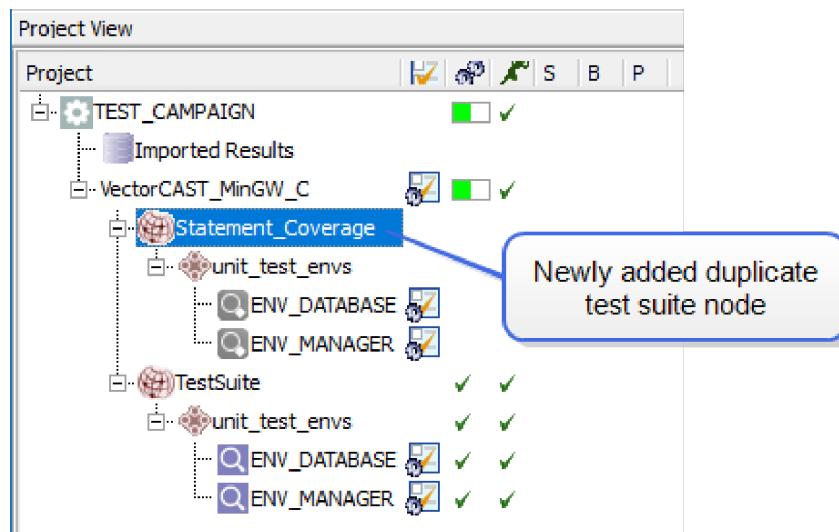


2. In the Duplicate dialog, give the new Test Suite the name **Statement_Coverage**.

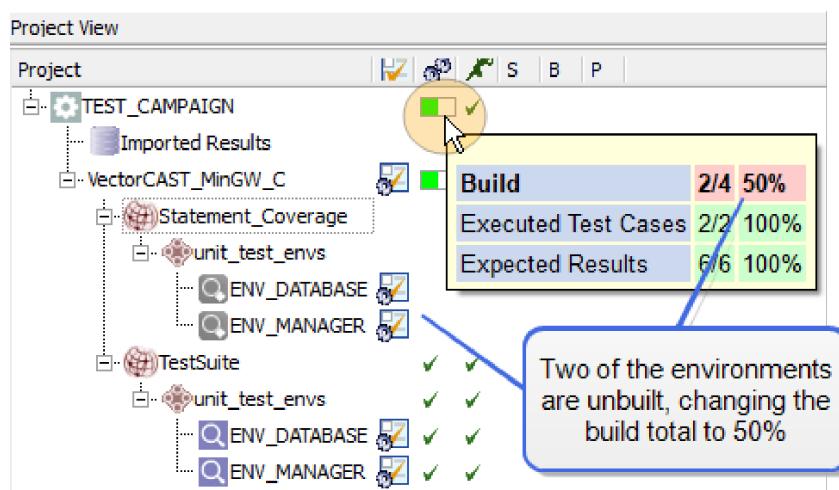


3. Click **Duplicate**.

A new Test Suite node is added to the project.

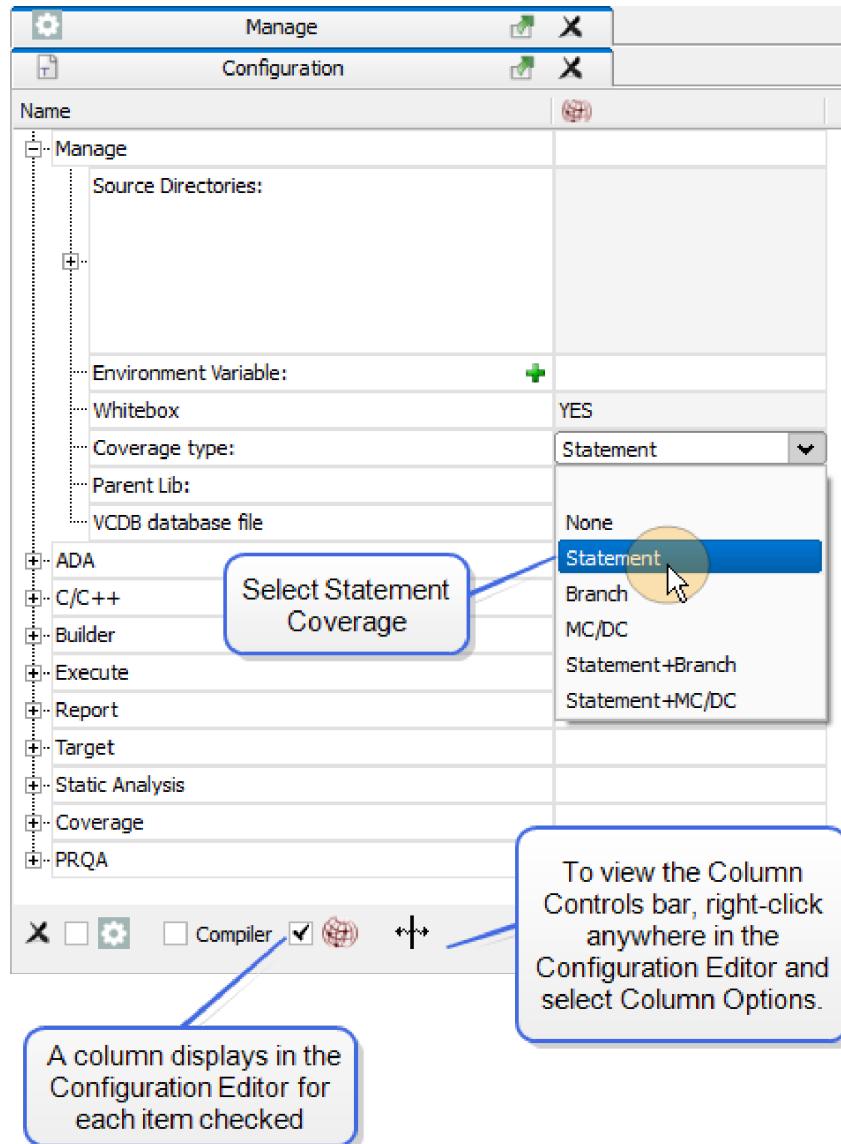


4. Hover your mouse over one of the percentage bars () in the Build column.

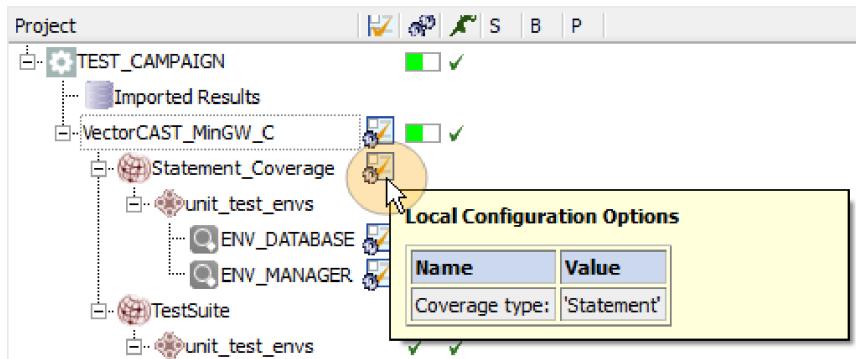


Now that a second Test Suite is present and has two unbuilt environments out of four total, the percentage of built/total is 50%, rather than 100%, as it was before. This information is reflected in the 50% green bar and the tooltip.

- Right-click the Test Suite named **Statement_Coverage** and choose **Open Configuration**. The Configuration Editor opens.

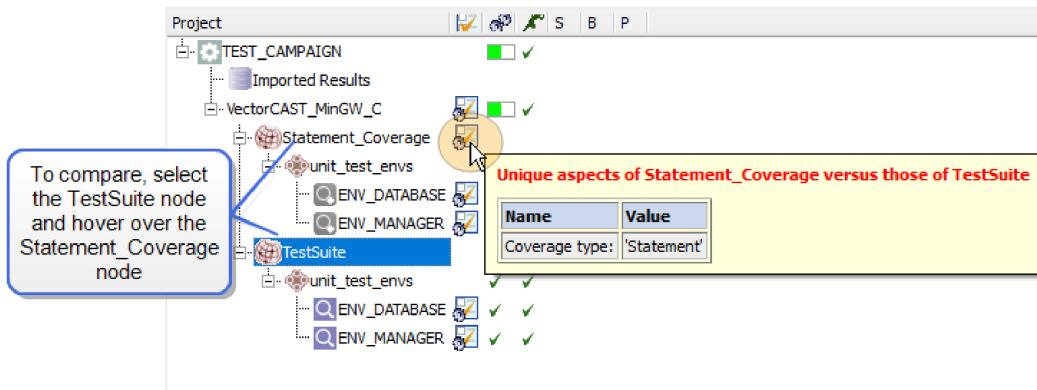


- In the Manage node, change the **Coverage type** to **STATEMENT**.
- Save the change by clicking the **Save** button on the Toolbar
- Close the Configuration Editor.
- Select the Test Suite **Statement_Coverage** and hover the mouse over the configuration icon.



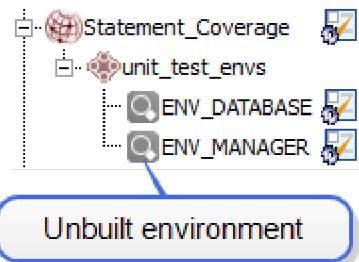
A tooltip for the Local Configuration Options shows you that the Coverage type has been changed to STATEMENT.

10. Select the TestSuite node and then hover the mouse over the configuration icon *for the other Test Suite, Statement_Coverage*.



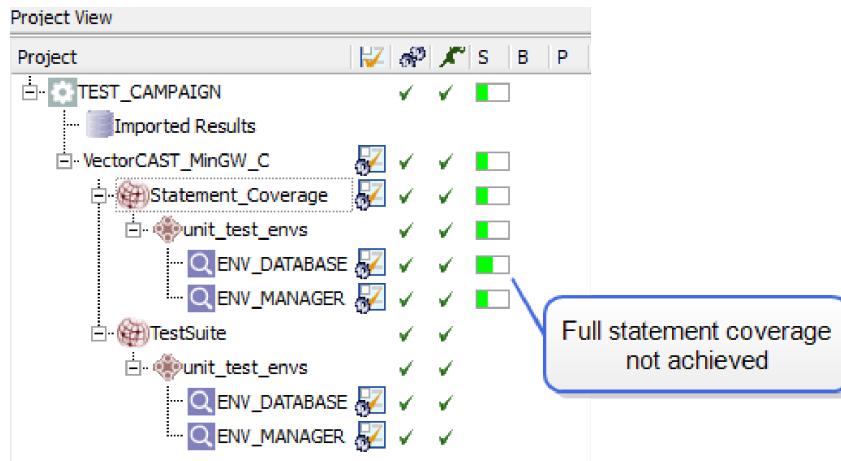
A tooltip shows you the differences between the two Test Suites. In our case, the coverage type differs between the two nodes.

11. Note that the two environments under the Statement_Coverage node are the same two environments found under the TestSuite node, but are represented with gray icons. This indicates they have not been built.



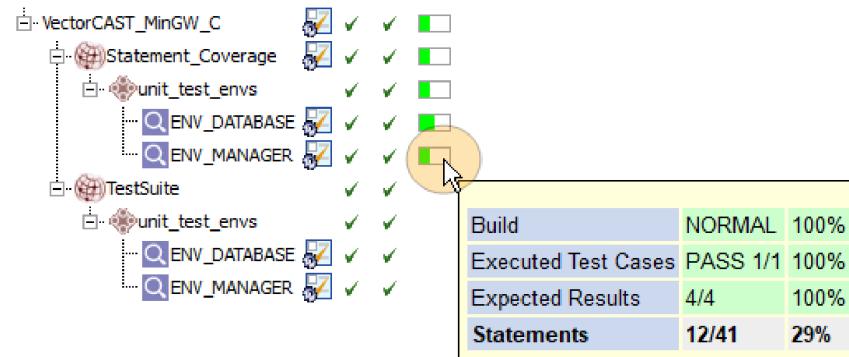
12. Right-click the Test Suite **Statement_Coverage** node and choose **Build/Execute => Full**.

VectorCAST builds the two environments using the settings from their parent Test Suite, **Statement_Coverage**, and then executes their test cases.



From the Status panel, you can see that both environments built successfully, both executed their test cases and they passed, but the status for Statement coverage shows only partial coverage.

13. Hover the mouse near the coverage status to see a tooltip with more detail.



The tooltip indicates:

- > Build: NORMAL – the Build status for the environment
- > Executed Test Cases: PASS – the Execution status for the environment because 1 out of 1 test cases passed.
- > Expected Results: 4 out of 4 Expected Value comparisons matched.
- > Statement: 29% of Statement coverage achieved

29% percent coverage is considered a failure because the default coverage percentage threshold is 100%. So in order to achieve a PASS status (green checkmark) for Statement coverage, the environment would need to have 100% coverage achieved or the threshold lowered.

14. Right-click on the TEST_CAMPAIGN node and choose **Reporting**.
15. The Manage Summary Report is displayed, showing the build, execution, and coverage status for the two environments in the different Test Suites. Note that one Test Suite has coverage and one

doesn't.

The screenshot shows the 'Manage Report' configuration data. It includes fields for report creation date (15 FEB 2018), time (12:09:03 PM), and version (18 (02/04/18)). Below is a 'Status Section' table:

	BUILD	BUILD TIME	EXPECTED	TESTCASES	EXECUTE TIME	Statements
ALL	4/4 (100%)	00:43	12/12 (100%)	4/4 (100%)	00:08	13/43 (30%)
VectorCAST_MinGW_C	4/4 (100%)	00:43	12/12 (100%)	4/4 (100%)	00:08	13/43 (30%)
Statement_Coverage	2/2 (100%)	00:24	6/6 (100%)	2/2 (100%)	00:04	13/43 (30%)
unit_test_envs	2/2 (100%)	00:24	6/6 (100%)	2/2 (100%)	00:04	13/43 (30%)
ENV_DATABASE	NORMAL	00:12	2/2 (100%)	1/1 (100%)	00:02	1/2 (50%)
ENV_MANAGER	NORMAL	00:12	4/4 (100%)	1/1 (100%)	00:02	12/41 (29%)
TestSuite	2/2 (100%)	00:19	6/6 (100%)	2/2 (100%)	00:04	-
unit_test_envs	2/2 (100%)	00:19	6/6 (100%)	2/2 (100%)	00:04	-
ENV_DATABASE	NORMAL	00:09	2/2 (100%)	1/1 (100%)	00:02	-
ENV_MANAGER	NORMAL	00:10	4/4 (100%)	1/1 (100%)	00:02	-

A callout box points to the 'TestSuite' row with the text 'There is no coverage for this test suite'.

Tutorial Summary

In this tutorial, you:

- > Created two unit test environments quickly, each having a test case with Expected Values.
- > Created a VectorCAST project and imported those environments.
- > Built and executed the environments and viewed a Summary report with information about the status.
- > Duplicated a Test Suite to apply coverage.
- > Built and executed the environments again, this time with coverage.
- > Noted the Status panel reflected the updated Test Execution status.

APPENDIX A: TUTORIAL SOURCE CODE LISTINGS

Requirements

FR11

Number of tables

The system will support 6 tables.

FR12

Number of seats per table

The system will support 4 seats per table.

FR13

List of entrees

The system will support the following entrees: steak chicken lobster pasta none

FR14

Placing an order updates occupied status

Placing an order updates the table's occupied status to true within the table database.

FR15

Placing an order updates number in party

Placing an order updates the table's number in party within the table database.

FR16

Placing an order updates a seat's order

Placing an order updates the seat's order within the table database.

FR17

Placing an order updates check total

Placing an order increases the table's check total within the table database, by an amount depending on the entree ordered, according to the following schedule: Entree: Amount steak: 14.0 chicken: 10.0 lobster: 18.0 pasta: 12.0 none: 0.0

FR18

Clearing a table resets occupied status

Clearing a table updates the table's occupied status to false within the table database.

FR19

Clearing a table resets number in party

Clearing a table updates the table's number in party to 0 within the table database.

FR20

Clearing a table resets orders for all seats

Clearing a table clears the orders for all seats of the table within the table database.

FR21

Clearing a table resets check total

Clearing a table updates the table's check total to 0.0 within the table database.

FR22

Obtaining check total

The system will provide a way to obtain the check total for a given table.

FR23

Size of waiting list

The system will support a waiting list of up to 10 parties.

FR24

Adding a party to waiting list

The system will provide a means of adding a party to the waiting list, with the party specified by name.

FR25

Getting the head of the waiting list

The system will provide a means of obtaining the name of the party at the head of the waiting list.

FR27

Adding free dessert

Placing certain orders will qualify the seat for free dessert, according to the following schedule:
Steak with caesar salad and a mixed drink qualifies a seat for pie. Lobster with green salad and wine
qualifies a seat for cake.

C**file: ctypes.h**

```

#ifndef _TUTORIAL_TYPES_H_
#define _TUTORIAL_TYPES_H_

#define SEATS_AT_ONE_TABLE 4
#define NUMBER_OF_TABLES 6

enum boolean { v_false, v_true };
enum soups { NO_SOUP, ONION, CHOWDER };
enum salads { NO_SALAD, CAESAR, GREEN };
enum entrees { NO_ENTREE, STEAK, CHICKEN, LOBSTER, PASTA };
enum desserts { NO_DESSERT, CAKE, PIE, FRUIT };
enum beverages { NO_BEVERAGE, WINE, BEER, MIXED_DRINK, SODA };

struct order_type
{
    enum soups     Soup;
    enum salads    Salad;
    enum entrees   Entree;
    enum desserts  Dessert;
    enum beverages Beverage;
};

typedef unsigned short seat_index_type;
typedef unsigned short table_index_type;

struct table_data_type
{
    enum boolean      Is_Occupied;
    seat_index_type   Number_In_Party;
    char             Designator;
    char             Wait_Person[10];
    struct order_type Order[SEATS_AT_ONE_TABLE];
    float            Check_Total;
};

typedef char name_type[32];

#endif /* _TUTORIAL_TYPES_H_ */

```

file: manager.c

```
#include "ctypes.h"
```

```

struct table_data_type Get_Table_Record(table_index_type Table);
void Update_Table_Record(table_index_type Table, struct table_data_type Data);

/* Allow 10 Parties to wait */
static name_type WaitingList[10];
static unsigned int WaitingListSize = 0;
static unsigned int WaitingListIndex = 0;

const struct order_type NULL_ORDER =
{NO_SOUP, NO_SALAD, NO_ENTREE, NO_DESSERT, NO_BEVERAGE};

/* This function will add a free dessert to specific orders based on the
   entree, salad, and beverage choice */
void Add_Included_Dessert(struct order_type* Order)
{
    if(Order->Entree == STEAK &&
       Order->Salad == CAESAR &&
       Order->Beverage == MIXED_DRINK) {

        Order->Dessert = PIE;

    } else if(Order->Entree == LOBSTER &&
              Order->Salad == GREEN &&
              Order->Beverage == WINE) {

        Order->Dessert = CAKE;
    }
}

int Place_Order(table_index_type Table,
               seat_index_type Seat,
               struct order_type Order)
{
    struct table_data_type Table_Data;

    Table_Data = Get_Table_Record(Table);

    Table_Data.Is_Occupied = v_true;
    Table_Data.Number_In_Party = Table_Data.Number_In_Party + 1;
    Table_Data.Order[Seat] = Order;

    /* Add a free dessert in some cases */
    Add_Included_Dessert(&Table_Data.Order[Seat]);
    switch(Order.Entree)
    {
        case NO_ENTREE :
            break;
    }
}

```

```

    case STEAK :
        Table_Data.Check_Total = Table_Data.Check_Total + 14.0;
        break;
    case CHICKEN :
        Table_Data.Check_Total = Table_Data.Check_Total + 10.0;
        break;
    case LOBSTER :
        Table_Data.Check_Total = Table_Data.Check_Total + 18.0;
        break;
    case PASTA :
        Table_Data.Check_Total = Table_Data.Check_Total + 12.0;
        break;
    }

    Update_Table_Record(Table, Table_Data);
    return 0;
}

int Clear_Table(table_index_type Table)
{
    struct table_data_type Table_Data;
    seat_index_type Seat;
    Table_Data = Get_Table_Record(Table);
    Table_Data.Is_Occupied = v_false;
    Table_Data.Number_In_Party = 1;

    for (Seat=0; Seat < SEATS_AT_ONE_TABLE; Seat++)
        Table_Data.Order[Seat] = NULL_ORDER;
    Table_Data.Check_Total = 0;
    Update_Table_Record(Table, Table_Data);
    return 0;
}

float Get_Check_Total(table_index_type Table)
{
    struct table_data_type Table_Data;
    Table_Data = Get_Table_Record(Table);
    return (Table_Data.Check_Total);
}

void Add_Party_To_Waiting_List(char* Name)
{
    int i = 0;
    if(WaitingListSize > 9)
        WaitingListSize = 0;

    while(Name && *Name) {
        WaitingList[WaitingListSize][i++] = *Name;
    }
}

```

```

        Name++;
    }

    WaitingList[WaitingListSize++][i] = 0;
}

char* Get_Next_Party_To_Be_Seated(void)
{
    if(WaitingListIndex > 9)
        WaitingListIndex = 0;
    return WaitingList[WaitingListIndex++];
}

```

file: database.c

```

#include "ctypes.h"

struct table_data_type Table_Data[NUMBER_OF_TABLES];

struct table_data_type Get_Table_Record(table_index_type Table)
{
    return (Table_Data[Table]);
}

void Update_Table_Record(table_index_type Table, struct table_data_type Data)
{
    Table_Data[Table] = Data;
}

```

file: whitebox.c

```

struct PointerType
{
    int DataIndex;
    int DataValue;
};

enum COLOR {RED, GREEN, BLUE};
enum DAY {MONDAY, TUESDAY, WEDNESDAY, THURSDAY};

static enum COLOR CurrentColor;
static enum DAY CurrentDay;

static struct PointerType P;

static void InitDay(enum DAY Val)
{
    CurrentDay = Val;
}

```

```

}

static void InitColor(enum COLOR Val)
{
    CurrentColor = Val;
}

void Initialize()
{
    InitDay(WEDNESDAY);
    InitColor(BLUE);
    P.DataIndex = 1;
    P.DataValue = 12;
}

```

file: manager_driver.c

```

#include <stdio.h>
#include <string.h>
#include "ctypes.h"

extern int Place_Order(table_index_type Table,
                      seat_index_type Seat,
                      struct order_type Order);

extern int Clear_Table(table_index_type Table);

extern float Get_Check_Total(table_index_type Table);

extern void Add_Included_Dessert(struct order_type* Order);

int main()
{
    struct order_type order;
    int Total;

    char line[10];

    printf("P=Place_Order C=ClearTable G=Get_Check_Total A=AddIncludedDessert : ");
    scanf("%s",line);

    switch (line[0])
    {
        case 'p': case 'P':
            order.Entree = STEAK;
            Place_Order(1, 1, order);
            break;
        case 'g': case 'G':

```

```

        order.Entree = CHICKEN;
        Place_Order(2, 2, order);
        Total = Get_Check_Total(2);
        printf("The Total is %d\n", Total);
        break;
    case 'c': case 'C':
        Clear_Table(1);
        break;
    case 'a': case 'A':
        order.Entree = STEAK;
        order.Salad = CAESAR;
        order.Beverage = MIXED_DRINK;
        Add_Included_Dessert(&order);
        break;
    }

    return 0;
}

```

C++**file: cpptypes.h**

```

#ifndef _TYPES_
#define _TYPES_

#if defined (__HC08__) || (defined (__HC12__) && defined (__PRODUCT_HICROSS_PLUS__))
typedef int bool;
#define false 0
#define true 1
#endif

const int SeatsAtOneTable = 4;
const int NumberOfTables = 6;

enum Soups {NoSoup, Onion, Chowder};
enum Salads {NoSalad, Caesar, Green};
enum Entrees {NoEntree, Steak, Chicken, Lobster, Pasta};
enum Desserts {NoDessert, Cake, Pie, Fruit};
enum Beverages {NoBeverage, Wine, Beer, MixedDrink, Soda};

struct OrderType
{
    enum Soups     Soup;
    enum Salads    Salad;
    enum Entrees   Entree;
    enum Desserts  Dessert;
    enum Beverages Beverage;

```

```

};

struct TableDataType
{
    bool      IsOccupied;
    int       NumberInParty;
    char      Designator;
    char      WaitPerson[10];
    OrderType Order[SeatsAtOneTable];
    int       CheckTotal;
};

typedef char name_type[32];

#endif

```

file: manager.h

```

#ifndef _MANAGER_
#define _MANAGER_

#include "cpptypes.h"
#include "database.h"

class Manager
{
public:
    Manager();
    void AddIncludedDessert(OrderType* Order);
    void PlaceOrder(int Table, int Seat, OrderType Order);
    void ClearTable(int Table);
    int GetCheckTotal(int Table);
    int MemberVariable;
    void AddPartyToWaitingList(char* Name);
    char* GetNextPartyToBeSeated(void);

private:
    DataBase Data;
    name_type WaitingList[10];
    unsigned int WaitingListSize;
    unsigned int WaitingListIndex;

};

#endif

```

file: manager.cpp

```
#include "cpptypes.h"
#include "database.h"
#include "manager.h"

Manager::Manager(){
    WaitingListSize = 0;
    WaitingListIndex = 0;
}

/* This function will add a free dessert to specific orders based on the
   entree, salad, and beverage choice */
void Manager::AddIncludedDessert(OrderType* Order)
{
    if(!Order)
        return;

    if(Order->Entree == Steak &&
       Order->Salad == Caesar &&
       Order->Beverage == MixedDrink) {

        Order->Dessert = Pie;

    } else if(Order->Entree == Lobster &&
              Order->Salad == Green &&
              Order->Beverage == Wine) {

        Order->Dessert = Cake;
    }
}

void Manager::PlaceOrder(int Table, int Seat, OrderType Order)
{
    TableDataType TableData;
    Data.DeleteTableRecord(&TableData);
    Data.GetTableRecord(Table, &TableData);

    TableData.IsOccupied = true;
    TableData.NumberInParty++;
    TableData.Order[Seat] = Order;
    /* Add a free dessert in some case */
    AddIncludedDessert(&TableData.Order[Seat]);

    switch(Order.Entree) {
        case Steak :
            TableData.CheckTotal += 14;
    }
}
```

```
        break;
    case Chicken :
        TableData.CheckTotal += 10;
        break;
    case Lobster :
        TableData.CheckTotal += 18;
        break;
    case Pasta :
        TableData.CheckTotal += 12;
        break;
    default :
        break;
    }
    Data.UpdateTableRecord(Table, &TableData);
}

void Manager::ClearTable(int Table)
{
    Data.DeleteRecord(Table);
}

int Manager::GetCheckTotal(int Table)
{
    TableDataType TableData;
    Data.DeleteTableRecord(&TableData);
    Data.GetTableRecord(Table, &TableData);
    return TableData.CheckTotal;
}

void Manager::AddPartyToWaitingList(char* Name)
{
    int i = 0;
    if(WaitingListSize > 9)
        WaitingListSize = 0;

    while(Name && *Name) {
        WaitingList[WaitingListSize][i++] = *Name;
        Name++;
    }
    WaitingList[WaitingListSize++][i] = 0;
}

char* Manager::GetNextPartyToBeSeated()
{
    if(WaitingListIndex > 9)
        WaitingListIndex = 0;
    return WaitingList[WaitingListIndex++];
}
```

file: database.h

```

#ifndef _DATABASE_
#define _DATABASE_

#include "cpptypes.h"

class DataBase
{
public :
    DataBase();           /* Constructor */
    ~DataBase();          /* Destructor */

    void GetTableRecord(int Table, TableDataType *Data);
    void UpdateTableRecord(int Table, TableDataType *Data);
    void DeleteRecord(int Table);

    void DeleteTableRecord(TableDataType *Data)
    {
        DeleteOneRecord(0, Data);
    }

private :
    void DeleteOneRecord(int Table, TableDataType *Data)
    {
        Data[Table].IsOccupied = false;
        Data[Table].NumberInParty = 0;
        Data[Table].Designator = ' ';
        Data[Table].WaitPerson[0] = '\0';
        Data[Table].CheckTotal = 0;

        for(int J=0;J<SeatsAtOneTable;J++)
        {
            Data[Table].Order[J].Soup      = NoSoup;
            Data[Table].Order[J].Salad     = NoSalad;
            Data[Table].Order[J].Entree    = NoEntree;
            Data[Table].Order[J].Dessert   = NoDessert;
            Data[Table].Order[J].Beverage  = NoBeverage;
        }
    }

    void DeleteAllRecords(int size, TableDataType *Data)
    {
        for(int I=0;I<size;I++)
            DeleteOneRecord(I, Data);
    }
};

```

```
#endif
```

file: database.cpp

```
#include "database.h"

TableDataType TableData[NumberOfTables];

DataBase:: DataBase() {}

DataBase::~ DataBase() {}

void DataBase::GetTableRecord(int Table, TableDataType* Data)
{
    *Data = TableData[Table];
}

void DataBase::UpdateTableRecord(int Table, TableDataType* Data)
{
    TableData[Table] = *Data;
}

void DataBase::DeleteRecord(int Table)
{
    DeleteOneRecord(Table, TableData);
}
```

file: whitebox.h

```
#ifndef _WHITEBOX_
#define _WHITEBOX_

struct PointerType
{
    int DataIndex;
    int DataValue;
};

class WhiteBox
{
public:
    WhiteBox() {}
    void Initialize();

private:
```

```

enum { RED, GREEN, BLUE };
enum { MONDAY, TUESDAY, WEDNESDAY, THURSDAY };
int CurrentColor;
int CurrentDay;
PointerType P;
void InitDay(int Val);
void InitColor(int Val);
};

#endif

```

file: whitebox.cpp

```

#include "whitebox.h"

void WhiteBox::Initialize()
{
    InitDay(MONDAY);
    InitColor(RED);
    P.DataIndex = 1;
    P.DataValue = 12;
}

void WhiteBox::InitDay(int Val)
{
    CurrentDay = Val;
}

void WhiteBox::InitColor(int Val)
{
    CurrentColor = Val;
}

```

file: manager_driver.cpp

```

#include "cpptypes.h"
#include "manager.h"
#include <stdio.h>

int main()
{
    OrderType order;
    Manager manager;
    int Total;

    char line[10];

```

```

printf("P=PlaceOrder C=ClearTable G=GetCheckTotal A=AddIncludedDessert : ");
scanf("%s",line);

switch (line[0])
{
    case 'p': case 'P':
        order.Entree = Steak;
        manager.PlaceOrder(1, 1, order);
        break;
    case 'g': case 'G':
        order.Entree = Chicken;
        manager.PlaceOrder(2, 2, order);
        Total = manager.GetCheckTotal(2);
        printf("The Total is %d\n", Total);
        break;
    case 'c': case 'C':
        manager.ClearTable(1);
        break;
    case 'a': case 'A':
        order.Entree = Steak;
        order.Salad = Caesar;
        order.Beverage = MixedDrink;
        manager.AddIncludedDessert(&order);
        break;
}
}

return 0;
}

```

file: Makefile

```

#--- Existing Makefile in this example doesn't contain any
Includes
INCLUDES =
CC = g++
source_files = manager.cpp database.cpp manager_driver.cpp
obj_files = $(patsubst %.cpp, %.o, $(source_files))
tutorial.exe : $(obj_files) $(source_files)
$(CC) -o tutorial.exe $(obj_files)
%.o : %.cpp
$(CC) -c $< $(INCLUDES) -o $@
all : tutorial.exe
clean :
rm -f tutorial.exe $(obj_files)

#####
##### Additions #####
##### for Wrap_Compiler command #####

```

```

# VCAST_COVER_ENV = CoverENV

## --- The Instrumented utility target below adds the Cover environment
## directory as an Include. The wrap_compile command then
## substitutes the compile command used in the existing Makefile.
## --- The object file dependency needed for linking the instrumented
## --- executable is then added to the existing set of object files.
## --- Lastly, the instrumented target lists each prerequisite target
## --- needed for the integration process. It sets the compiler
## --- template, compiles the source files, creates the object file
## --- dependency, and creates the instrumented executable.

# instrumented : INCLUDES += -I $(VCAST_COVER_ENV)
# instrumented : CC = $(VECTORCAST_DIR)/clicast -e $(VCAST_COVER_ENV) Cover Source wrap_
compile statement g++
# instrumented : obj_files += c_cover_io.o
# instrumented : VCAST_CCAST_CFG $(obj_files) c_cover_io.o all

##### The c_cover_io.o target below creates the object file needed to
##### link the instrumented executable. The target needs to include
##### vcast_c_options.h as a prerequisite in the case that the coverage
##### type is changed. The c_cover_io.cpp file will need to be
##### recompiled to reflect the change to the coverage type.
# c_cover_io.o : $(VCAST_COVER_ENV)/c_cover_io.cpp $(VCAST_COVER_ENV)/vcast_c_options.h
#g++ -c $< $(INCLUDES) -o $@

# ----- Set the Compiler Template -----
# VCAST_CCAST_CFG :
#$ (VECTORCAST_DIR)/clicast -lc template GNU_CPP_34

#####
# uninstrumented : obj_files += c_cover_io.o
# uninstrumented :
#$ (VECTORCAST_DIR)/clicast -e $(VCAST_COVER_ENV) Cover
Instrument None
#rm -f tutorial.exe TESTINSS.DAT $(obj_files)
#rm -r $(VCAST_COVER_ENV)*
# .PHONY : clean instrumented uninstrumented

```

Ada

file: types.adb

```
package TYPES is
```

```

SEATS_AT_ONE_TABLE : constant integer := 4;
NUMBER_OF_TABLES   : constant integer := 6;

type SOUPS      is (NO_ORDER, ONION, CHOWDER);
type SALADS     is (NO_ORDER, CAESAR, GREEN);
type ENTREES    is (NO_ORDER, STEAK, CHICKEN, LOBSTER, PASTA);
type DESSERTS   is (NO_ORDER, CAKE, PIE, FRUIT);
type BEVERAGES  is (NO_ORDER, WINE, BEER, MIXED_DRINK, SODA);

type ORDER_TYPE is
  record
    SOUP      : SOUPS := NO_ORDER;
    SALAD     : SALADS := NO_ORDER;
    ENTREE    : ENTREES := NO_ORDER;
    DESSERT   : DESSERTS := NO_ORDER;
    BEVERAGE  : BEVERAGES := NO_ORDER;
  end record;

type SEAT_RANGE_TYPE is new integer range 0 .. SEATS_AT_ONE_TABLE;
subtype SEAT_INDEX_TYPE is SEAT_RANGE_TYPE range 1 .. SEAT_RANGE_TYPE'last;
type TABLE_INDEX_TYPE is new integer range 1..NUMBER_OF_TABLES;
subtype WAIT_PERSON_NAME is string (1..10);

type TABLE_ORDER_TYPE is array (SEAT_INDEX_TYPE) of ORDER_TYPE;

type TABLE_DATA_TYPE is
  record
    IS_OCCUPIED      : boolean := false;
    NUMBER_IN_PARTY  : SEAT_RANGE_TYPE := 0;
    DESIGNATOR       : character := ' ';
    WAIT_PERSON      : WAIT_PERSON_NAME := "          ";
    ORDER            : TABLE_ORDER_TYPE;
    CHECK_TOTAL      : integer := 0;
  end record;

end TYPES;

```

file: manager.ads

```

with TYPES;

package MANAGER is

procedure PLACE_ORDER (
  TABLE : in      TYPES.TABLE_INDEX_TYPE;
  SEAT  : in      TYPES.SEAT_INDEX_TYPE;
  ORDER : in out  TYPES.ORDER_TYPE);

```

```

procedure ADD_INCLUDED_DESSERT ( ORDER : in out TYPES.ORDER_TYPE );

procedure CLEAR_TABLE (
    TABLE : in      TYPES.TABLE_INDEX_TYPE);

function GET_CHECK_TOTAL (TABLE : TYPES.TABLE_INDEX_TYPE)
    return integer;

procedure ADD_PARTY_TO_WAITING_LIST ( NAME : in      string );

function GET_NEXT_PARTY_TO_BE_SEATED return string;

end MANAGER;

```

file: manager.adb

```

with DATABASE;
with TYPES; use TYPES;

package body MANAGER is

    MAX_NAME_LENGTH : constant := 20;
    subtype WAITING_LIST_NAME_TYPE is string(1..MAX_NAME_LENGTH);
    type WAITING_LIST_TYPE;
    type WAITING_LIST_TYPE_PTR is access WAITING_LIST_TYPE;
    type WAITING_LIST_TYPE is
        record
            NAME : WAITING_LIST_NAME_TYPE := ( others => ' ' );
            NEXT : WAITING_LIST_TYPE_PTR := null;
        end record;
    WAITING_LIST : WAITING_LIST_TYPE_PTR := null;

    procedure PLACE_ORDER (
        TABLE : in      TYPES.TABLE_INDEX_TYPE;
        SEAT  : in      TYPES.SEAT_INDEX_TYPE;
        ORDER : in out TYPES.ORDER_TYPE) is

        TABLE_DATA : TYPES.TABLE_DATA_TYPE;

    begin

        DATABASE.GET_TABLE_RECORD (
            TABLE => TABLE,
            DATA  => TABLE_DATA);

        TABLE_DATA.IS_OCCUPIED := true;
        TABLE_DATA.NUMBER_IN_PARTY := TABLE_DATA.NUMBER_IN_PARTY + 1;
        TABLE_DATA.ORDER (SEAT) := ORDER;

```

```

ADD_INCLUDED_DESSERT ( TABLE_DATA.ORDER(SEAT) );

case ORDER.ENTREE is
when TYPES.NO_ORDER =>
    null;
when TYPES.STEAK =>
    TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 14;
when TYPES.CHICKEN =>
    TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 10;
when TYPES.LOBSTER =>
    TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 18;
when TYPES.PASTA =>
    TABLE_DATA.CHECK_TOTAL := TABLE_DATA.CHECK_TOTAL + 12;
end case;

DATABASE.UPDATE_TABLE_RECORD (
    TABLE => TABLE,
    DATA  => TABLE_DATA);

end PLACE_ORDER;

procedure ADD_INCLUDED_DESSERT ( ORDER : in out TYPES.ORDER_TYPE ) is
begin
    if ORDER.ENTREE      = STEAK and
        ORDER.SALAD       = CAESAR and
        ORDER.BEVERAGE   = MIXED_DRINK
    then
        ORDER.DESSERT := PIE;
    elsif ORDER.ENTREE     = LOBSTER and
          ORDER.SALAD      = GREEN and
          ORDER.BEVERAGE  = WINE
    then
        ORDER.DESSERT := CAKE;
    end if;
end ADD_INCLUDED_DESSERT;

procedure CLEAR_TABLE (
    TABLE : in      TYPES.TABLE_INDEX_TYPE) is

TABLE_DATA : TYPES.TABLE_DATA_TYPE;
NULL_ORDER : TYPES.ORDER_TYPE :=
    (SOUP      => TYPES.NO_ORDER,
     SALAD     => TYPES.NO_ORDER,
     ENTREE    => TYPES.NO_ORDER,
     DESSERT   => TYPES.NO_ORDER,
     BEVERAGE  => TYPES.NO_ORDER);

```

```

begin

    DATABASE.GET_TABLE_RECORD (
        TABLE => TABLE,
        DATA => TABLE_DATA);

    TABLE_DATA.IS_OCCUPIED := false;
    TABLE_DATA.NUMBER_IN_PARTY := 1;
    for SEAT in TYPES.SEAT_INDEX_TYPE loop
        TABLE_DATA.ORDER (SEAT) := NULL_ORDER;
    end loop;
    TABLE_DATA.CHECK_TOTAL := 0;

    DATABASE.UPDATE_TABLE_RECORD (
        TABLE => TABLE,
        DATA => TABLE_DATA);

end CLEAR_TABLE;

function GET_CHECK_TOTAL (TABLE : TYPES.TABLE_INDEX_TYPE)
    return integer is

    TABLE_DATA : TYPES.TABLE_DATA_TYPE;

begin

    DATABASE.GET_TABLE_RECORD (
        TABLE => TABLE,
        DATA => TABLE_DATA);

    return TABLE_DATA.CHECK_TOTAL;

end GET_CHECK_TOTAL;

procedure ADD_PARTY_TO_WAITING_LIST (NAME : in string) is
    TO_ADD : WAITING_LIST_TYPE_PTR := new WAITING_LIST_TYPE;
begin
    if NAME'length > MAX_NAME_LENGTH then
        TO_ADD.NAME := NAME(NAME'first..NAME'first+MAX_NAME_LENGTH-1);
    else
        TO_ADD.NAME(1..NAME'length) := NAME;
    end if;
    TO_ADD.NEXT := WAITING_LIST;
    WAITING_LIST := TO_ADD;
end ADD_PARTY_TO_WAITING_LIST;

function GET_NEXT_PARTY_TO_BE_SEATED return string is

```

```

PTR : WAITING_LIST_TYPE_PTR := WAITING_LIST;
RET : WAITING_LIST_NAME_TYPE := ( others => ' ' );
begin
  if PTR = null then
    null;
  elsif PTR.NEXT = null then
    RET := PTR.NAME;
    WAITING_LIST := null;
  else
    while PTR.NEXT.NEXT /= null loop
      PTR := PTR.NEXT;
    end loop;
    RET := PTR.NEXT.NAME;
    PTR.NEXT := null;
  end if;
  for IDX in reverse 1 .. RET'length loop
    if RET(IDX) /= ' ' then
      return RET(1..IDX);
    end if;
  end loop;
  return "";
end GET_NEXT_PARTY_TO_BE_SEATED;

end MANAGER;

```

file: database.ads

```

with types;

package DATABASE is

  procedure GET_TABLE_RECORD (
    TABLE : in      TYPES.TABLE_INDEX_TYPE;
    DATA  : out     TYPES.TABLE_DATA_TYPE);

  procedure UPDATE_TABLE_RECORD (
    TABLE : in      TYPES.TABLE_INDEX_TYPE;
    DATA  : in      TYPES.TABLE_DATA_TYPE);

end DATABASE;

```

file: database.adb

```

with types;

package body DATABASE is

```

```

TABLE_DATA : array (TYPES.TABLE_INDEX_TYPE) of TYPES.TABLE_DATA_TYPE;

procedure GET_TABLE_RECORD (
    TABLE : in      TYPES.TABLE_INDEX_TYPE;
    DATA  : out     TYPES.TABLE_DATA_TYPE) is

begin
    DATA := TABLE_DATA (TABLE);
end GET_TABLE_RECORD;

procedure UPDATE_TABLE_RECORD (
    TABLE : in      TYPES.TABLE_INDEX_TYPE;
    DATA  : in      TYPES.TABLE_DATA_TYPE) is

begin
    TABLE_DATA (TABLE) := DATA;
end UPDATE_TABLE_RECORD;

end DATABASE;

```

file: manager_driver.adb

```

with TEXT_IO;

with TYPES;
with MANAGER;
procedure manager_driver is
    CHOICE : string(1..10);
    LEN    : integer := 0;
    ORDER  : TYPES.ORDER_TYPE;
begin
    TEXT_IO.PUT("P=Place_Order C=ClearTable G=Get_Check_Total A=AddIncludedDessert : ");
    TEXT_IO.GET_LINE(CHOICE,LEN);

    if LEN > 0 then
        case CHOICE(1) is
            when 'p' | 'P' =>
                ORDER.ENTREE := TYPES.STEAK;
                MANAGER.PLACE_ORDER ( 1, 1, ORDER );
            when 'g' | 'G' =>
                ORDER.ENTREE := TYPES.CHICKEN;
                MANAGER.PLACE_ORDER ( 2, 2, ORDER );
                LEN := MANAGER.GET_CHECK_TOTAL ( 2 );
                TEXT_IO.PUT_LINE ( "The Total is" & integer'image ( LEN ) );
            when 'c' | 'C' =>

```

```

MANAGER.CLEAR_TABLE ( 1 );

when 'a' | 'A' =>
    ORDER.ENTREE    := TYPES.STEAK;
    ORDER.SALAD     := TYPES.CAESAR;
    ORDER.BEVERAGE  := TYPES.MIXED_DRINK;
    MANAGER.ADD_INCLUDED_DESSERT ( ORDER );

when others =>
    null;

end case;

end if;

end manager_driver;

```

file: whitebox.ads

```

package WHITEBOX is

    type pointer_type is private;

    procedure initialize (pointer : in out pointer_type);

private
    type pointer_type is
        record
            data_index : integer;
            data_value : integer;
        end record;

end WHITEBOX;

```

file: whitebox.adb

```

package body WHITEBOX is

    type color is (red, green, blue);
    type day   is (monday, tuesday, wednesday, thursday);

    procedure init_day (val : day);

    procedure init_color (val : color);

    current_day : day;
    current_color : color;

```

```

procedure initialize (pointer : in out pointer_type) is
begin
    init_day (day'first);
    init_color (color'first);
    pointer.data_index := 1;
    pointer.data_value := 12;
end initialize;

procedure init_day (val : day) is
begin
    current_day := val;
end init_day;

procedure init_color (val : color) is
begin
    current_color := val;
end init_color;

end WHITEBOX;

```

Build Settings

file: types.h

```

#ifndef TYPES_H
#define TYPES_H

enum SkyType
{
    Cloudy,
    MostlyCloudy,
    PartlyCloudy,
    MostlySunny,
    Sunny
};

typedef enum SkyType ESkyType;

enum TemperatureType
{
    Hot,
    Warm,
    Mild,
    Cool,
    Cold
};

typedef enum TemperatureType ETemperatureType;

```

```
#endif
```

file: sky.h

```
#ifndef SKY_H
#define SKY_H

#include "types.h"

ESkyType sky_condition( int cloud_coverage );

const char* sky_description( ESkyType s );

#endif
```

file: sky.c

```
#include "sky.h"

ESkyType sky_condition( int cloud_coverage )
{
    if ( cloud_coverage >= 95 )
        return Cloudy;
    else if ( cloud_coverage >= 70 )
        return MostlyCloudy;
    else if ( cloud_coverage >= 30 )
        return PartlyCloudy;
    else if ( cloud_coverage >= 5 )
        return MostlySunny;
    else
        return Sunny;
}

const char* sky_description( ESkyType s )
{
    const char* description = "Unknown";
    switch ( s )
    {
        case Cloudy:
            description = "Cloudy";
            break;
        case MostlyCloudy:
            description = "Mostly Cloudy";
            break;
        case PartlyCloudy:
            description = "Partly Cloudy";
            break;
    }
}
```

```

        case MostlySunny:
            description = "Mostly Sunny";
            break;
        case Sunny:
            description = "Sunny";
            break;
    }
    return description;
}

```

file: temperature.h

```

#ifndef TEMPERATURE_H
#define TEMPERATURE_H

#include "types.h"

ETemperatureType temperature_condition_fahrenheit( int degrees_fahrenheit );

const char* temperature_description( ETemperatureType s );

#ifdef CELSIUS
ETemperatureType temperature_condition_celsius( int degrees_celsius );
int celsius_to_fahrenheit( int celsius );
#endif

#endif

```

file: temperature.c

```

#include "temperature.h"

ETemperatureType temperature_condition_fahrenheit( int degrees_fahrenheit )
{
    if ( degrees_fahrenheit >= 80 )
        return Hot;
    else if ( degrees_fahrenheit >= 70 )
        return Warm;
    else if ( degrees_fahrenheit >= 60 )
        return Mild;
    else if ( degrees_fahrenheit >= 40 )
        return Cool;
    else
        return Cold;
}

const char* temperature_description( ETemperatureType t )

```

```
{
    const char* description = "Unknown";
    switch ( t )
    {
        case Hot:
            description = "Hot";
            break;
        case Warm:
            description = "Warm";
            break;
        case Mild:
            description = "Mild";
            break;
        case Cool:
            description = "Cool";
            break;
        case Cold:
            description = "Cold";
            break;
    }
    return description;
}

#ifndef CELSIUS
ETemperatureType temperature_condition_celsius( int degrees_celsius )
{
    return temperature_condition_fahrenheit( celsius_to_fahrenheit( degrees_celsius ) );
}

int celsius_to_fahrenheit( int celsius )
{
    int fahrenheit = (celsius * 9 / 5) + 32;
    return fahrenheit;
}
#endif
```

file: weather.c

```
#include <stdio.h>
#include <stdlib.h>

#include "temperature.h"
#include "sky.h"

#ifndef CELSIUS
#define TEMPERATURE_SCALE "Celsius"
#elif defined(FAHRENHEIT)
#define TEMPERATURE_SCALE "Fahrenheit"
```

```
#endif

int main( int argc, char **argv )
{
    ETemperatureType temperature_cond;
    ESkyType sky;
    const char* temperature_desc;
    const char* sky_desc;

    if ( argc != 3 )
    {
        printf( "USAGE: weather [temperature in degrees %s] [percentage of cloud coverage]
\n", TEMPERATURE_SCALE );
        return 1;
    }

#define CELSIUS
    temperature_cond = temperature_condition_celsius( atoi( argv[1] ) );
#define FARENHEIT
    temperature_cond = temperature_condition_fahrenheit( atoi( argv[1] ) );
#endif
    temperature_desc = temperature_description( temperature_cond );

    sky = sky_condition( atoi( argv[2] ) );
    sky_desc = sky_description( sky );

    printf( "The weather is %s and %s.\n", temperature_desc, sky_desc );
    return 0;
}
```

file: Makefile

```
INC_DIR=inc
SRC_DIR=src
HEADERS=$(INC_DIR)/types.h $(INC_DIR)/temperature.h $(INC_DIR)/sky.h
OBJECTS=weather.o temperature.o sky.o
DEGREE_TYPE=-DCELSIUS

weather: $(OBJECTS)
$(CC) -o weather weather.o temperature.o sky.o

weather.o: $(SRC_DIR)/weather.c $(HEADERS)
$(CC) -c $(SRC_DIR)/weather.c $(DEGREE_TYPE) -Iinc

temperature.o: $(SRC_DIR)/temperature.c $(HEADERS)
$(CC) -c $(SRC_DIR)/temperature.c $(DEGREE_TYPE) -Iinc

sky.o: $(SRC_DIR)/sky.c $(HEADERS)
```

```
$(CC) -c $(SRC_DIR)/sky.c -Iinc  
  
clean:  
rm -f weather.exe $(OBJECTS)
```

file: Makefile.nmake

```
CC=cl.exe  
INC_DIR=inc  
SRC_DIR=src  
HEADERS=$(INC_DIR)/types.h $(INC_DIR)/temperature.h $(INC_DIR)/sky.h  
OBJECTS=weather.obj temperature.obj sky.obj  
DEGREE_TYPE=CELSIUS  
  
weather.exe: $(OBJECTS)  
$(CC) weather.obj temperature.obj sky.obj  
  
weather.obj: $(SRC_DIR)/weather.c $(HEADERS)  
$(CC) /c $(SRC_DIR)/weather.c /D$(DEGREE_TYPE) /Iinc  
  
temperature.obj: $(SRC_DIR)/temperature.c $(HEADERS)  
$(CC) /c $(SRC_DIR)/temperature.c /D$(DEGREE_TYPE) /Iinc  
  
sky.obj: $(SRC_DIR)/sky.c $(HEADERS)  
$(CC) /c $(SRC_DIR)/sky.c /Iinc  
  
clean:  
del weather.exe $(OBJECTS)
```

Index

- actual results 15
- building compound test cases 72, 79
- building test cases 14, 39, 110, 183
- building the compound test case environment 69
- c tutorial
 - introduction 23
- class
 - testable
 - meaning of 17
 - closure concept 18
 - ignoring units 19
 - minimizing 18
 - code coverage
 - about 52, 127, 197
 - using 52, 127, 197
 - compiler node
 - definition of 293
 - components
 - of test harness 16
 - compound test case
 - building 72, 79
 - definition 69, 145, 207
 - environment
 - building 69
 - Coverage menu
 - Initialize
 - Statement 52, 128, 197
 - creating reports 15
 - data persistence
 - definition 69, 145, 207
 - data source
 - definition of 293
 - demonstration 240
 - dependent unit
 - meaning of 17
 - smart stubs 17
 - directories
 - search in project 297
 - driver program
 - meaning of 16
 - enterprise testing
 - description of 292
 - how it works 292
 - environment
 - building in Enterprise Testing 302
 - constructor 14
 - creating 24, 69, 94, 173
 - executing in Enterprise Testing 303
 - import source 297
 - meaning of 16
 - environment build
 - for import to project 295
 - Environment menu
 - Update Environment 70
 - View
 - Test Case Management Report 51, 126, 196
 - environment variables
 - FLEXLM_NO_CKOUT_INSTALL_LIC 9
 - LM_APP_DISABLE_CACHE_READ 9
 - LM_LICENSE_FILE 9
 - VECTOR_LICENSE_FILE 9
 - environment view 39, 110, 183
 - event
 - defined 17
 - examples
 - included with VectorCAST 20
 - executing test cases 15, 49, 123, 194

expected results
 adding to a test case 43, 45, 115, 117, 188, 190

factoring
 compiler options 298
 environment options 298
 no options 298
 search directories 298

File menu
 Close 68, 144, 206
 Print 51, 127, 197
 Recent Environments 69
 Save As 42, 114, 187

files
 cpptypes.h 318
 ctypes.h 313
 database.adb 331
 database.ads 331
 database.c 316
 database.cpp 323
 database.h 322
 enterprise_unit_testing.zip 293
 Makefile 325, 338
 Makefile.nmake 339
 manager.adb 328
 manager.ads 327
 manager.c 313
 manager.cpp 320
 manager.h 319
 manager_driver.adb 332
 manager_driver.c 317
 manager_driver.cpp 324
 sky.c 335
 sky.h 335
 temperature.c 336
 temperature.h 336
 tutorial 20

 tutorial_report.html 51, 127, 197
 types.ads 326
 types.h 334
 used in tutorials 20
 weather.c 337
 whitebox.adb 333
 whitebox.ads 333
 whitebox.c 316
 whitebox.cpp 324
 whitebox.h 323
 global objects 14
 group
 definition of 292
 help
 importing examples 11
 importing examples 11
 .csv file 11
 examples.csv 11
 input values 14
 integration testing 69, 145, 207
 Lint
 integration 273
 tutorial 273
 main window
 resizing 68, 144, 206
 multi UUT tutorial 69, 145, 207
 options
 factoring 297
 output values 14
 overview 13
 overview of coverage process 15
 printing test results 51, 127, 197
 project
 creating 293
 definition of 292

project tree
 description of 300

prototype stubbing
 meaning of 17

registry
 disallow modifications to 9
 disallow reading from 9
 modifying with license settings 8
 reading license settings from 9

regression testing
 catching a bug with 293

reports
 coverage summary 15
 creating 15
 execution history 15
 in Enterprise Testing 304
 summary of results 15
 test data 15

repository
 creating a 295

right-click menu
 Compound Only 74
 Execute 49, 123, 194
 Expand All Array Indices 76
 Insert Test Case 74, 110
 on COMPOUND 79
 Rename 40, 111, 184
 saving test results to a file 51, 127, 197

SBF
 meaning of 17

simple test case
 definition 69, 145, 207

smart stubs 17

starting VectorCAST 8

static analysis
 lint 273

status panel
 description of 301

stub-by-function
 enabling 57

Stub by function
 meaning of 17

stubs
 meaning of 17

test case
 meaning of 17

test cases
 building 14, 39, 110, 183
 building summary 43, 115, 188
 executing 15, 49, 80, 123, 194
 stubbing by function 57

test driver
 meaning of 17

test harness 37, 108, 181
 components 16
 meaning of 16

Test menu
 View
 Test Case Data 43, 49, 114, 122, 187, 194

test results
 printing 51, 127, 197
 saving 51, 127, 197

test suite
 definition of 292

testable class
 meaning of 17

toolbar
 Execute button 49, 123, 194
 New button 26, 232
 Print button 51, 127, 197
 Save button 42, 49, 114, 122, 187, 194

troubleshooting

- don't see expected values 67, 143, 205
- don't see Get_Table_Record 65
- tutorial
 - building test cases 39, 110, 183
 - building the environment 24, 94, 173
 - entering expected values 190
 - entering input values 44-45, 116-117, 189-190
 - introduction 19
 - lint 273
 - multi UUT 69, 145, 207
 - overview 19
 - source code listings
 - C 313
 - C++ 318
 - summary of 65, 91, 141, 170, 204, 229, 238, 255, 270, 282, 290, 310
 - test case building summary 43, 115, 188
 - troubleshooting 65, 67, 143, 205
- tutorial program 240, 255
- tutorial program Ada 172
- tutorial program C++ 23, 93

Unit Under Test

- meaning of 16

USER_GLOBALS_VCAST

- about 40, 111, 184

UUT

- meaning of 16

VectorCAST

- starting 8

VectorCAST/Cover

- demonstration 240
- tutorial 240, 255

visible subprogram

- meaning of 17

Welcome

- importing examples 11

Welcome page 9

Whitebox Test Environment

- building 231

Whitebox Tutorial

- building a test environment 231

Wizard

- VectorCAST Project 295