Vector Software

WHITEPAPER

Using C++ with Test Driven Development

Introduction

This whitepaper explains how Test Automation tools can be used with C++ to support Test Driven Development (TDD) in an Agile-programming environment. This paper assumes some basic familiarity with Test Automation products.

Traditional Software Development

Most projects currently use a test automation tool in a traditional development environment where individual C++ classes are unit tested as they are developed. This pure unit testing allows the developer to ensure that the low-level requirements attributed to a unit are implemented correctly, while at the same time verifying the completeness of the testing via code coverage analysis.

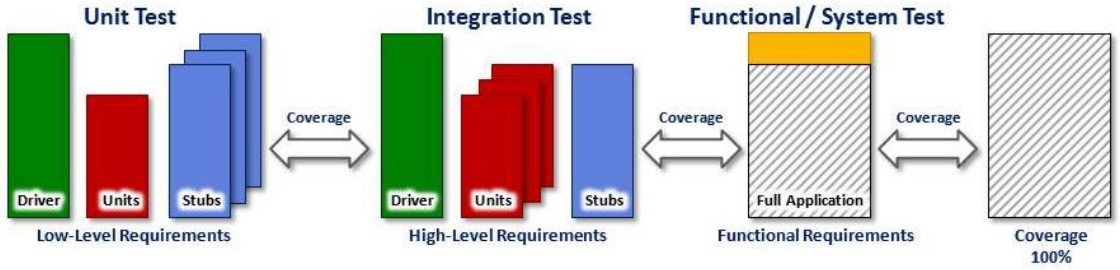

Figure 1 - Traditional Software Development - Waterfall Methodology

Since the code has already been written at this point, there are a number of issues in working this way:

- *The identification of a potential defect in the code developed is delayed by several days, if not weeks, after it has been written*
- *There is a risk that developers will create unit tests based on the code developed, rather than deriving them directly from the requirements*
- *Code is over engineered, and more code is developed than is required to satisfy the project requirements*

All of these issues result in identifying potential issues with the code later and later into the project life cycle. It is well known that the cost of fixing a defect increases the longer the period of time between it being introduced, and it finally being detected and corrected (See Figure 2).

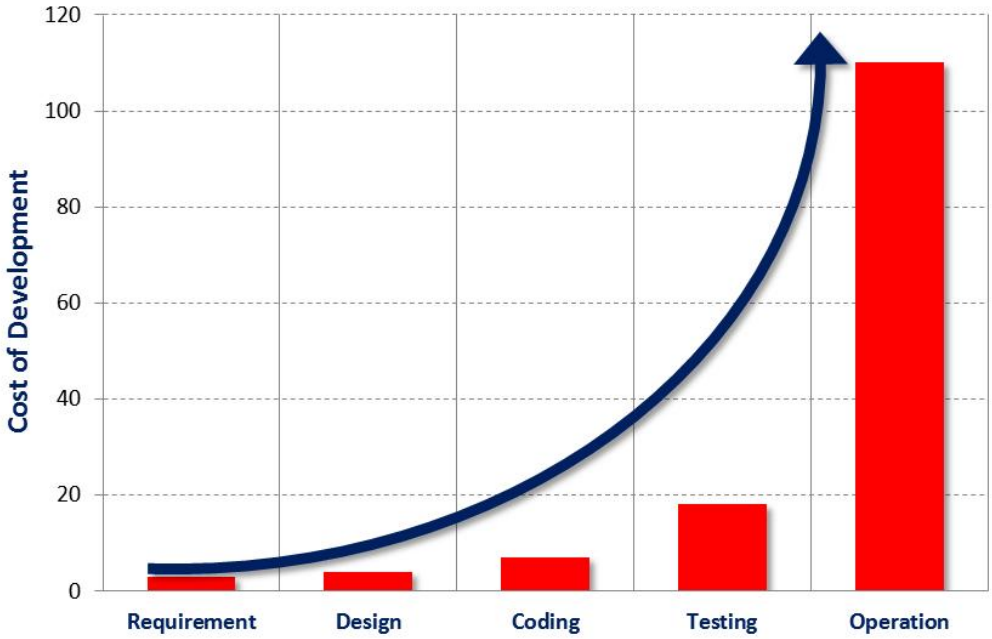

Figure 2 - Cost of Fixing Defects in the Software Development Lifecycle

A common solution to this is to introduce code inspections or code static analysis. However, code inspection is expensive and can become complicated when verifying complex algorithms or system behaviors. On the other hand, static analysis only verifies there are no ambiguities in the code written, rather than the code that is actually correct.

Additionally, there are some key points to consider.

1. The scope of most errors is fairly limited. *"Eighty percent of the errors are found in twenty percent of a project's classes or routines."* (Endres 1975, Gremillion 1984, Boehm 1987b, Shull et al 2002).

2. Most common errors are the programmer's fault. *"Ninety-five percent are caused by programmers, two percent by systems software (i.e.; compiler and OS), two percent by some other software, and one percent by the hardware."* (McConnell, Steve. Code Complete. Redmond: Microsoft Press, 2004)

Therefore, we can draw the conclusion, the sooner we test the programmers software, the sooner we can find the majority of the errors in the system.

Test Driven Development

Test Driven Development (TDD) aims to solve this problem by moving test case development earlier in the process to a point after the design is created, but before the code is written. TDD is a software development process that relies on the repetition of a very short development cycle. This encourages simple designs and inspires confidence much earlier in the software lifecycle.

The concepts of TDD are simple and can be broken into a 3 step process:

1. The developer writes a test case to verify a desired improvement or new function. This test case will fail until the functionality is correctly implemented
2. The developer then produces code to pass that test
3. Once the code is passing the developer refactors the new code to acceptable standards

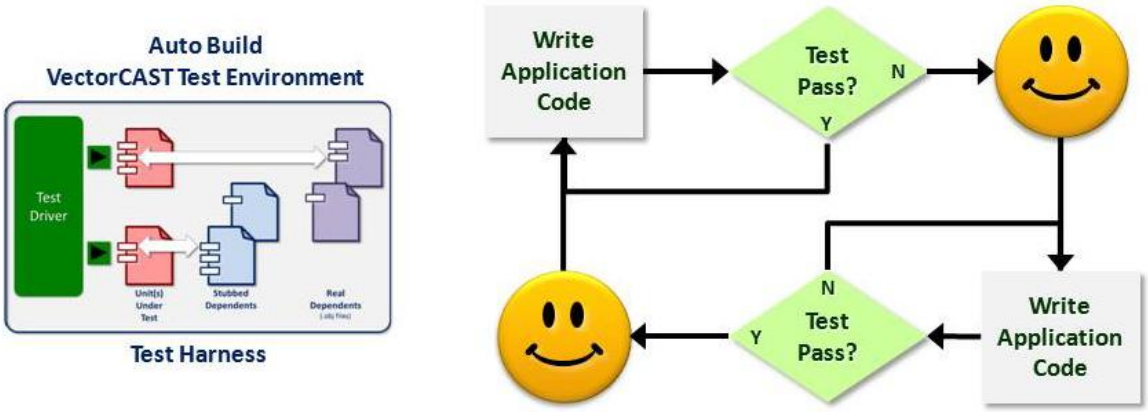

Figure 3 - Test Driven Development workflow

These concepts originally derive from test-first programming which is a core part of extreme programming that originated in 1992. However, in more recent years Test Driven Development has become a more general topic in its own right.

This could be further simplified into what is called the 3 Laws of Test Driven Development.

1. You may not write production code until you have written a failing unit test
2. You may not write more of a unit test than is sufficient to fail and not compiling is failing
3. You may not write more production code than is sufficient to pass the currently failing test

Using C++ with Test Driven Development

The primary difference between using C++ in "Test-Driven Development" mode versus "Traditional Unit Testing" is that for TDD mode, the input to the test environment construction is simply a group of C++ header files (.h files). There is no requirement for the source files.

When you are building a test environment, you simply point to the C++ header files that you want to create test cases for and then choose one or more header files as the Unit(s) Under Test. Your test automation tool will create the test environment. As there are no source files available initially, it will create empty source files for the methods inside the Unit(s) Under Test. This results in a complete executable test harness that can be run. At this point all other features of your test automation should tool work exactly the same way that they do in "Traditional" mode.

Key Features in a Test Automation tool to make TDD possible

Some characteristics of a Test Automation tool will make adaptability to Test Driven Development Processes easier.

Traceability between test cases and source code

One of the key concepts of TDD is to be able to correlate a test case with the exact lines of coded executed. This allows the developer to verify that only the code associated with proving that Test cases have been executed, and nothing more. The easiest way to achieve this is through code coverage, where the tool can link the code coverage collected directly to the test cases executed (see figure 4).

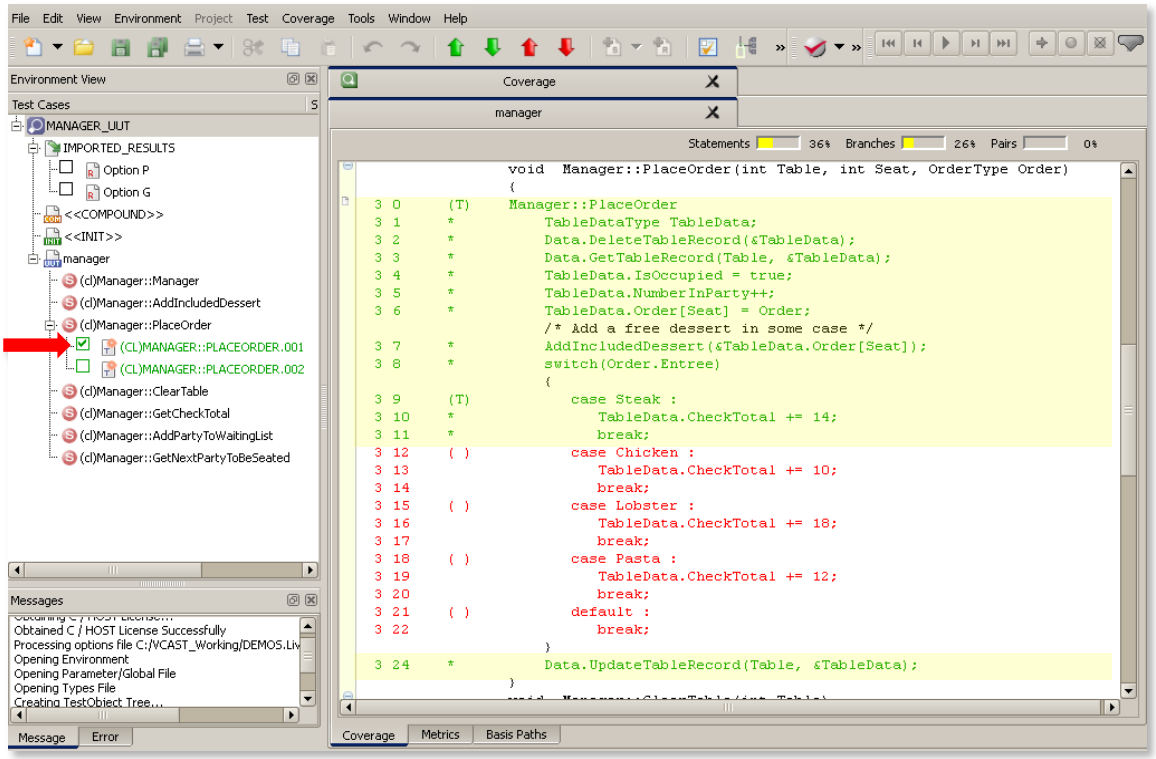

Figure 4 - Traceability between a test case and the underlying source code tested

Traceability between test cases and requirements

As test cases must be built before the underlying code is developed, the ability to link the Test cases to the requirements used to derive them can ensure that there is a direct traceability between the test cases, the requirements, and the code that is verified. This ensures that we stay within the 3 Laws of TDD, and will make the process of refactoring our code much easier (see Figure 5).

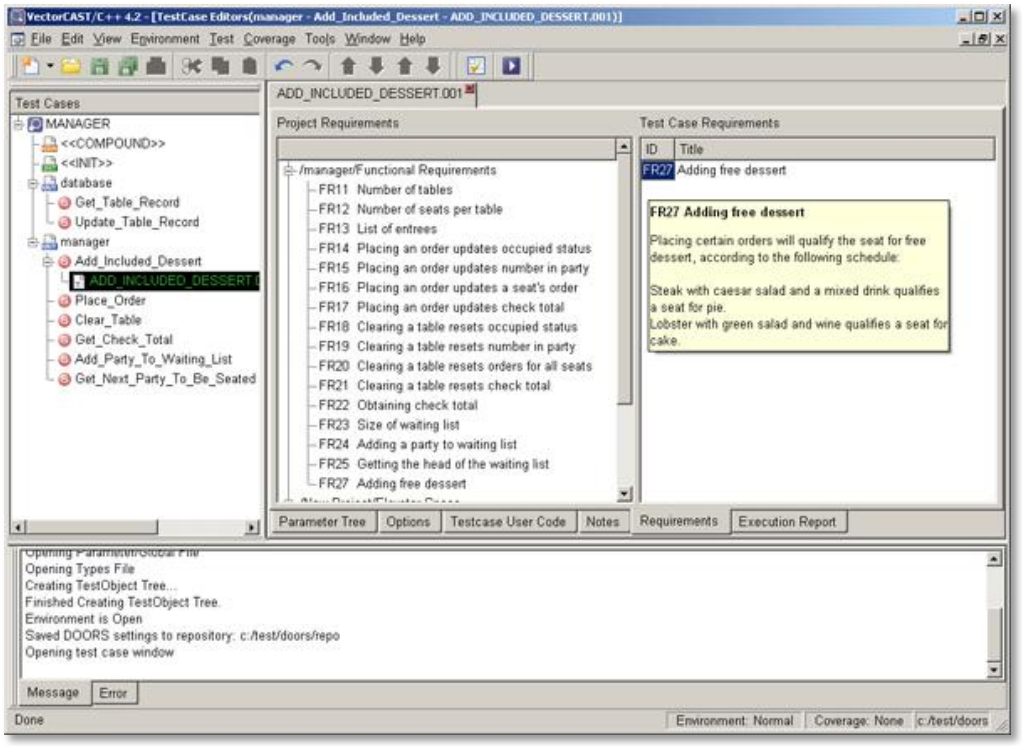

Figure 5 - Linking Requirements to Test Cases

A Test Driven Development Example

Assume that we are building a new application, and one of the modules will provide message processing. We are not sure of the complete functionality needed, but we are sure that we need to be able to send and receive messages, and further, that the messages will consist of a variable number of 32 bit integer values. We will first create a simple C++ header file and define a procedure to send messages, and a procedure to receive messages. It might look like the following:

```
// Message.h
bool Send_Message(
            MessageIdType  Message_Id,
            ByteSteam &MessageData,
            MessageSizeType MessageSize);

bool Receive_Message(
            MessageIdType  &Message_Id,
            ByteSteam &MessageData,
            MessageSizeType &MessageSize);
```

Now rather than go forward with the implementation details of these procedures as the next step, Test Driven Development requires us to generate test cases for these two procedures. The great thing about using this approach is it gets us to think about the design details and the edge conditions before we start to write the code. For example, here are some of the test cases that should be created for this module:

- *When you call Receive_Message, do you get the message back in the order that they were sent or is there a priority to the messages based on ID?*
- *What happens when we receive and there are no pending messages?*
- *What is the largest size of a message that can be sent? Is it in the most appropriate data type for the Message_Size?*
- *What is the range of Message_Id values?*
- *What happens if an invalid Message_Id is received?*

The answers to all of these questions can be decided and using a Test Automation tool, test cases can be developed for them prior to writing any code. The test cases that you initially build will fail, but they will contain the inputs and expected outputs for the functions that formalize the expected behavior.

The implementation of the functions can be built incrementally. As the code is built, the already existing test cases will start to pass, and eventually, when the code is complete, all of the test cases should pass.

About Vector Software

Vector Software, Inc., is the leading independent provider of automated software testing tools for developers of safety critical embedded applications. Vector Software's VectorCAST line of products, automate and manage the complex tasks associated with unit, integration, and system level testing. VectorCAST products support the C, C++, and Ada programming languages.

Vector Software, Inc.
1351 South County Trail, Suite 310
East Greenwich, RI 02818 USA
T: 401 398 7185
F: 401 398 7186
E: info@vectorcast.com

Vector Software
33 Glasshouse Street, Suite 3.08
London W1B 5DG, UK
T: +44 203 178 6149
F: +44 20 7022 1651
E: info@vectorcast.com

Vector Software
Vorster Straße 80
47906 Kempen Germany
T: +49 2152 8088808
F: +49 2152 8088888
E: info@vectorcast.com