

Unit Level Software Testing: *Extra Effort Upfront Saves Time and Boosts Safety*

by *David S. Bernazzani*
Intertech Engineering Associates, Inc.

as published in *Today's Medical Developments*,
March, 2007



A unit test strategy thought about during the development phase will assist the developers in structuring its testability, as well as make the code more reliable during the remaining phases of testing.

A System-level, or “black-box”, testing verifies that software correctly implements the system-level requirements and specifications. It requires no knowledge of the software design or structural implementation. In contrast, unit-level testing is based on detailed knowledge of the architectural and logical design, data structures, performance and timing specifications, and interface requirements. The result is software that does what it is supposed to do and that is far less likely to fail, to require costly rework, or to result in harm to patients.

Regulators recognize this. The FDA, in its General Principles of Software Validation guidance document, recommends that device software “should be challenged with test cases based on its internal structure, and with test cases based on its external specifications.”

What Is Unit Testing?

White-box testing is based on internal structure. Unit testing is a form of whitebox testing that is done at or near the code level to ensure that the implementation matches the intended design. The design must be documented in sufficient detail so that tests can

be developed to verify the accuracy of the software implementation. The idea is to test that small sections of code (typical functions) work as intended and are reasonably robust. Once these individual code blocks are tested, larger groups of those building blocks will hold up better during integration.

The execution of a unit test should be as fully automated as possible and should be given strict pass/fail criteria based on the intended design. Unit test configurations often allow batching or scripting of tests to be run. The

COMPANY PROFILE

Intertech Engineering Associates, Inc.

Address: 100 Lowder Brook Avenue
Suite 2500
Westwood, MA 02090
www.inea.com - (781) 801-1100

Industry: (Electro)Medical Devices

Services: Assessments
Training
Consulting
Hands-on Engineering

Skills: Product Design
Risk Management
Requirements Engineering
Electronics Development
Software Development
Software Verification and Validation
Production/Quality System Software Validation

scripts allows automated loading of the units to be tested, feeding of test inputs, capture of test outputs, and logging of pass/fail results.

This allows the unit tests to be run quickly and the results to be tabulated with great efficiency. There should be little left open to interpretation by the tester. Unit testing that requires tester interaction (such as pressing buttons or stepping through the code with a debugger) should only be used as a last resort, but may be necessary for some areas (e.g. critical startup routines that may be coded in assembly). A high level of automation allows the developer to exercise the code after a build, but prior to release, for integration or systems-level testing. This would not be practical if the unit testing was by manual means. Frequently re-running the tests provides some confidence that the code was not “broken” in some major way with the most recent changes. Stopping errors at this early release stage can save time as the code goes through the integration process.

What Is a Unit?

From a regulatory perspective, the device manufacturer can define a unit to be any grouping of source code. However, the definition of a unit for unit testing generally should be done at the smallest, most practical code level.

A unit can be defined as a single function, group of related functions, or an entire file of functions to verify the code is in relative isolation. The most traditional approach is to define a unit to be at the function level ensuring that each function behaves properly and the design was implemented correctly. In cases where there are groups of related functions, testing can be done on the set of functions and related data. For classes, one possible decision is to apply unit tests to every public method, since the private methods will inherently be covered as part of the object-oriented hierarchy. By testing at the higher levels and stubbing out fewer internal functions, code can easily be refactored by the developer, without wholesale changes to the unit tests that cover them. Care should be given not to test at too high a level, it is possible to lose some control if the

testing tries to encompass too many functions at once.

Tools, Techniques and Methodologies

Unit tests should be written to verify that the code is a thorough and accurate implementation of the design that is described in the design documentation. The assumption that the design elements of the documentation have been traced back to the requirements is implicit. This assures that the design being tested is actually one that the user and other product stakeholders need. The core algorithms, computations and path coverage should be tested for compliance with the design documentation as much as possible.

The unit test strategy should be planned as part of the development phase. Creating code that contains a framework for unit testing is easier than trying to shoehorn unit testing in later. Finding errors during unit testing is also more cost effective to the project, rather than waiting for problems to be found in systems tests, which take longer to develop, debug, run and process.

The unit test framework is usually straightforward and requires the ability to isolate code, drive testing and report results. The framework code handles communicating with the PC to download test input parameters and to upload test results. Output is often captured to a file for rapid analysis. An expected-results file is critical for objectively determining if the unit has behaved as expected.

The results should include the output of the test invocation, as well as a log of each sub-unit that was invoked. For object-oriented systems, it is possible to test one object by using mock objects to intelligently fake the various interfaces in the system, leaving the tested object free to run under controlled circumstances. Sets of mock-object interfaces can be interchanged with real objects as part of integration testing.

By controlling the inputs and checking the resulting sequence of function calls and outputs, a unit can be verified to work under a wide range of conditions. Both in-range and out-of-range inputs should be tried, this includes minimum values, maximum values, boundary values and any corner cases that may occur as a result of errant software. Programmers often assume that the

calling functions will never pass in incorrect parameters. However, software systems are complex, and ensuring that a function is robust and defensive often helps when other, more complicated areas of software cause invalid values to be propagated within the system.

Several tool suites (JUnit, VJUnit, CPPUnit, Cantata++) can be used as the test framework. It is even possible to create a custom framework for testing as part of code development. The unit being tested must be compiled using the same compiler and toolset as the actual product software.

Additionally, the testing should run on the actual hardware and processor to validate the compiled code. If leaving unit test hooks in place is a problem, or if code space is tight, compilation switches can be used to switch the tests off for actual releases.

How Much Should Be Tested?

Generally, 80% of the errors are found in 20% of the code. Spend the most time on functions containing key algorithms, safety, are complex or are likely to be defective. How do you know what is likely to be defective? Large functions, complex functions, code with multiple branches, complex calculations or deep layers of nesting are good targets. Functions written by less experienced or less skilled engineers should also be considered for more focused testing. Straight-line code is unlikely to turn up much in the way of problems and is not likely to be broken by changes to the code. Spend the most time computations, looping constructs and areas with data structure access.

Unit testing typically finds problems with corner cases, lack of input checking and computational problems, flaws that might not be easily detected at a systems level. Both expected and unexpected inputs should be tried, focusing especially on the boundary conditions that are most likely to occur in the case of errant software. Core algorithms that drive the software are easily stressed when controlling the inputs as part of unit testing.

Unit Testing and Defensive Programming

Unit testing helps ensure that code is robust. It indirectly strengthens segments of code by exposing improper

handling of unexpected failures that crop up in the software. Resolution of these findings results in software in which unexpected failures are properly trapped and corrected before the device fails and harms the patient.

Consequently, unit tests generally make individual units more defensive and less reliant on checks in other areas of software. If bugs in other areas of software are introduced during later development, this defensive posture will make an excellent safeguard. This tends to aid in the debugging process as well, since robust code will be more likely trap with a meaningful error rather than, say, hung code followed by a watchdog reset.

Unit testing, however, will generally not catch integration errors. The focus here is on the smaller building blocks in the system to ensure that each does its job properly. How these smaller pieces interact is not likely to be found with unit testing, which is why integration testing and systems-level testing is so critical to the entire validation process.

Summary

Unit testing is best thought about during development, not afterwards. Creating a unit-test strategy will help the developers structure the code to be more easily testable. This leads to more robust code and code that is generally less tightly coupled (i.e. unit more independent and easier to unit test) than code written without unit testing in mind. Unit testing should be automated as much as possible so that regression is as simple as building and running the code.

This will save time in later phases of the development effort as the software iteration process occurs. For best results, an independent person should review the test and update for proper coverage. The FDA wants independent review and testing of the device software as much as possible. Finally, don't be leery of the unit-test process. Proper unit testing done early will make the code far more reliable as the software enters the remaining phases of testing. A little more time up front will pay dividends down the road.

ABOUT THE AUTHOR:



David Vogel is the founder and president of Intertech Engineering Associates, Inc.

Dr. Vogel was a participant in a joint AAMI/FDA workgroup to develop a standard for Critical Device Software Validation which was subsequently included in the IEC 62304 Software Lifecycle Standard. He was also a participant on

the joint AAMI/FDA workgroup to develop a Technical Information Report (TIR) for Medical Device Software Risk Management. Currently, Dr. Vogel is a member of the AAMI/FDA workgroup developing a TIR on Quality System Software Validation.

A frequent lecturer for workshops and seminars on topics related to medical device development and validation, Dr. Vogel also is the author of numerous publications and holds several patents.

Dr. Vogel received a bachelor's degree in electrical engineering from Massachusetts Institute of Technology. He earned a master's degree in biomedical engineering, a master's degree in electrical and computer engineering, and a doctorate in biomedical engineering from the University of Michigan.

Intertech Service Offerings:

Risk Analysis and Management
Software Design and Development
Electronic Design and Development
Requirements Development and Management
Documentation and Traceability
Verification and Validation
Evaluations, Reviews, Inspections
Planning
Project Management
Compliance Consulting and Training
Manufacturing and Quality System Software Validation

Leverage INTERTECH's expertise to:

Reduce Project Risk

Shorten Time to Market

Cut Development and Test Cost

Assure Quality Products

ABOUT INTERTECH:

Intertech Engineering Associates has been helping medical device manufacturers bring their products to market since 1982. Through a distinct top-down service model, Intertech offers high-level consulting and hands-on engineering. By balancing technical expertise and practical business experience, we support clients through all phases of product development. While we do make your job easier, Intertech exists not to replace but to partner with clients to help balance the concerns of quality, time and cost.

With considerable experience in FDA regulatory compliance, our time-tested development process can anticipate and solve problems inexpensively on the planning board rather than through costly solutions later in the development, test, or post-deployment phases. By using deliberate processes, Intertech ensures an improvement in quality and can build client expertise.

Call us today for more information or a free consultation at 781.801.1100

INTERTECH Engineering Associates, Inc.

100 Lowder Brook Drive Suite 2500 Westwood, MA 02090 USA

www.inea.com - info@inea.com - Tel: (781) 801-1100 - Fax: (781) 801-1108