Robustness Testing of Software-Intensive Systems: Explanation and Guide

Julie Cohen
Dan Plakosh
Kristi Keeler

April 2005

Acquisition Support Program

Technical Note
CMU/SEI-2005-TN-015

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

List of Tables

Acknowledgments

The authors would like to acknowledge the technical guidance and assistance provided by the SEI video department during this project, especially John Antonucci and Dan Bidwa.

Abstract

Many Department of Defense (DoD) programs engage in what has been called “happy-path testing” (that is, testing that is only meant to show that the system meets its functional requirements). While testing to ensure that requirements are met is necessary, often tests aimed at ensuring that the system handles errors and failures appropriately are neglected. Robustness has been defined by the Food and Drug Administration as “the degree to which a software system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.” This technical note provides guidance and procedures for performing robustness testing as part of DoD or federal acquisition programs that have a software component. It includes background on the need for robustness testing and describes how robustness testing fits into DoD acquisition, including source selection issues, development issues, and developmental and operational testing issues.

1 Introduction

1.1 Purpose

This technical note provides guidelines for performing robustness testing as part of Department of Defense (DoD) or federal acquisition programs that have a software component. These tests can be done as part of a source selection demonstration or as part of a software or system test program during almost any test phase.

Details of the test procedures are provided mainly for Windows-based systems, but most of the test procedures can also be used on applications that run on other operating systems. Specific robustness tests for embedded real-time systems are not included in this report.

1.2 Why Consider Robustness Testing?

During several Independent Technical Assessments (ITAs),[1] the Carnegie Mellon® Software Engineering Institute (SEI) found that some programs were testing software to only the minimum extent to ensure that the required functionality was met. Very limited or no testing was performed to ensure that the system could handle unexpected user input or system failures. The government program offices often didn't realize that this type of "happy-path" testing left the system open to vulnerabilities that might not surface until much later in the development cycle or after deployment.

The value of detecting errors early in the development cycle, both in terms of cost and schedule, is well known. Boehm and Basili reported [Boehm 91], "Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase."

This statement is as true for robustness problems as it is for functionality defects. The sooner these types of errors can be found and corrected, the less impact there is on the program cost and schedule. If these types of problems aren't discovered during system testing, they can manifest themselves during operations in ways that can be very difficult to diagnose and fix, disrupting operations and causing very expensive troubleshooting and repair activities.

[1] An ITA is an objective, technical evaluation of a program, typically conducted due to cost, schedule, or performance problems. An ITA sponsor is typically a Component Acquisition Executive (CAE), Program Executive Officer (PEO), or System Program Director (SPD). An ITA typically examines both the program office and contractor(s), focuses on software-intensive system development or acquisition, and addresses the *entire* program scope—planning, design, implementation, testing, and maintenance.

One other advantage to the robustness testing concepts addressed in this technical note is that much of this testing can be performed on both commercial off-the-shelf (COTS) software and custom software. It is often difficult to determine the quality of COTS products, and this type of robustness testing can at least provide a top-level indication of the ability of COTS products to detect and handle situations that can produce errors.

1.3 Definitions

Unless otherwise noted, the definitions provided in this section are from the ApTest Web site (http://www.aptest.com) [ApTest 03].

Testing:

- the process of exercising software to verify that it satisfies specified requirements and to detect errors
- the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item [IEEE 98a]
- the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component

Boris Beizer identified five attitudes toward testing [Beizer 90]. These attitudes, which are described below, can affect the way a person or a project will perform testing.

- Phase 0: There's no difference between testing and debugging. Other than in support of debugging, testing has no purpose.
- Phase 1: The purpose of testing is to show that the software works.
- Phase 2: The purpose of testing is to show the software doesn't work.
- Phase 3: The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable level.
- Phase 4: Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.

The testing described in this report is at the Phase 2-3 level.

Robustness testing is known by many different names. Many of these are equivalent, and some are used to define a specific type of robustness testing. Many of these terms are defined below.

Ad hoc testing: a testing phase where the tester tries to "break" the system by randomly trying the system's functionality. This can include negative testing.

Boundary value analysis/testing: an analysis that focuses on "corner cases" or values that are usually out of range as defined by the specification. This means that if a function expects all values in the range of negative 100 to positive 1000, test inputs would include negative 101 and positive 1001.

Compatibility testing: testing whether software is compatible with other elements of a system with which it should operate (e.g., browsers, operating systems, or hardware)

Endurance testing: checks for memory leaks or other problems that may occur with prolonged execution

End-to-end testing: testing a complete application environment in a situation that mimics real-world use (such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems, if appropriate)

Exhaustive testing: testing that covers all combinations of input values and preconditions for an element of the software under test

Exploratory testing: Exploratory testing has several uses: the one that is germane to robustness testing is to provide rapid feedback on a product's quality on short notice, with little time, without advance thought, or early in development when the system may not be stable [Copeland 04].

Negative testing: testing aimed at showing that software does not work. This is also known as "test to fail" or "dirty testing."

> The British Standard (BS) definition of negative testing in BS 7925-1 [BCS 98] is taken from Beizer [Beizer 90] and defines negative testing as "testing aimed at showing software does not work." Lyndsay pointed out [Lyndsay 03], "This can lead to a range of complementary and competing aims:
>
> - discovery of faults that result in significant failures: crashes, corruption, and security breaches
> - observation and measurement of a system's response to external problems
> - exposure of software weakness and potential for exploitation"
>
> Lyndsay also expanded this definition to include tests for

- input validation, rejection, and re-requesting functionality (human input and external systems)
- internal data validation and rejection
- coping with absent, slow, or broken external resources
- error-handling functionality (i.e., messaging, logging, monitoring)
- recovery functionality (i.e., fail-over, rollback, and restoration)

Recovery testing: testing to confirm that the program recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power-out conditions.

Scalability testing: performance testing focused on ensuring the application under test gracefully handles increases in workload

Security testing: testing to confirm that the program can restrict access to authorized personnel and that the authorized personnel can access the functions available to their security level

Soak testing: running a system at high load for a prolonged period of time (for example, running several times more transactions in an entire day [or night] than would be expected in a busy day) to identify any performance problems that appear after a large number of transactions have been executed

Stress testing: testing conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how it fails

1.4 Contents of This Report

This report is structured as follows:

- Section 1 provides general background and definitions.
- Section 2 provides information on testing guidance in relation to robustness testing. In addition, DoD, federal, and commercial testing sources are discussed.
- Section 3 describes the actual robustness testing procedures.
- Section 4 describes how robustness testing fits into DoD acquisition, including source selection issues, development issues, as well as developmental and operational testing issues.
- Appendix A provides a table of the specific tests called out in Section 3.
- Appendix B contains additional references to software testing in commercial standards.

- Appendix C contains links to resources that may be useful when dealing with robustness testing.
- Appendix D contains an acronym list.

2 Software Testing Guidance

2.1 DoD Policy

DoD Instruction 5000.2 discusses two types of testing: one type is accomplished during the development phase and the other during operational test and evaluation (OT&E). Developmental testing is typically performed by the contractor during system development, and OT&E is typically carried out by the service using the production system. There has been a trend over the past several years to consider combining developmental and operational testing where feasible.

2.1.1 Developmental Testing

DoD Directive 5000.1 does not specifically address developmental testing (often referred to as developmental test and evaluation [DT&E]). It addresses the overall Integrated Test and Evaluation, which includes DT&E, Operational Test and Evaluation, and Live Fire Testing as follows:

> Integrated Test and Evaluation. Test and evaluation shall be integrated throughout the defense acquisition process. Test and evaluation shall be structured to provide essential information to decision-makers, assess attainment of technical performance parameters, and determine whether systems are operationally effective, suitable, survivable, and safe for intended use. The conduct of test and evaluation, integrated with modeling and simulation, shall facilitate learning, assess technology maturity and interoperability, facilitate integration into fielded forces, and confirm performance against documented capability needs and adversary capabilities as described in the system threat assessment.

DoD Directive 5000.2 adds the following DT&E specific information to this definition: “A well planned and executed DT&E program supports the acquisition strategy and the systems engineering process, providing the information necessary for informed decision making throughout the development process and at each acquisition milestone. DT is the verification and validation of the systems engineering process and must provide confidence that the system design solution is on track to satisfy the desired capabilities.”

2.1.2 OT&E

The requirements for OT&E originate in U.S. law. Title 10 of the U.S. Code discusses operational test and evaluation, stating that the Director of OT&E must determine "whether the results of such test and evaluation confirm that the items or components actually tested are effective and suitable for combat." DoD Directive 5000.2 also discusses operational test and evaluation: "Operational test and evaluation shall determine the effectiveness and suitability of the system."

2.2 Service-Specific Test Requirements

Each service implements the policies outlined in DoD 5000 with their own set of instructions. The following sections briefly describe the service-specific policies for the Air Force, the Army, and the Navy.

2.2.1 Air Force Test Policy

The Air Force test policy is contained in Air Force Instruction (AFI) 99-101, *Developmental Test and Evaluation,* for developmental test and in AFI 99-102, *Operational Test and Evaluation,* for OT&E. There are no specific references to robustness testing in either of these documents, and the specific sections covering software testing are relatively brief. The software section in AFI 99-101 states, "During DT&E, testers will set quantitative and demonstrable performance objectives for software, and structure the testing to demonstrate software has reached a level of maturity appropriate for each phase." Under the paragraph on System Compatibility and Interoperability, AFI 99-101 does state that the test should "evaluate network hardware and software system effectiveness, suitability, interoperability, compatibility, and integration." Robustness testing would certainly fall under this umbrella.

AFI 99-102 addresses at least a portion of the motivation for robustness testing, that of stress testing. The instruction states, "OT&E will be conducted in as realistic an operational environment as possible and practical, and to identify and help resolve deficiencies as early as possible. These conditions must be representative of both combat stress and peacetime operational conditions."

There are many other Air Force (AF) resources for software testing, including

- AF Pamphlet 99-116, *Test and Evaluation Management Guide*
- AFI 33-108, *Compatibility, Interoperability, and Integration of Command, Control, Communications, Computer (C4) Systems*

2.2.2 Army Test Policy

U.S. Army Test Policy is contained in Army Regulation 73-1, *Test and Evaluation Policy*. This regulation covers both developmental and operational testing. Developmental testing is defined as follows: "Developmental test is a generic term encompassing [Modeling and Simulation] M&S and engineering type tests that are used to verify that design risks are minimized, that system safety is certified, that achievement of system technical performance is substantiated, and that readiness for OT&E is certified." Robustness testing could fall under the heading of ensuring that the design risks are minimized.

Operational testing is defined as follows: "The [operational test] OT is a field test of a system or item under realistic operational conditions with users who represent those expected to operate and maintain the system when it is fielded or deployed." Again, while there is no specific reference to robustness types of testing, at least parts of robustness testing could be justified as testing under realistic operational conditions.

The Army also includes independent test as an additional testing category. The regulation states, "The primary objective of the independent system evaluation is to address the demonstrated effectiveness, suitability, and survivability of Army and multi-Service systems for use by typical users in realistic operational environments. The primary objective of the independent system assessment is to address system potential effectiveness, suitability, and survivability." Both developmental and operational testing can be sources for independent test activities.

Other Army documents providing test guidance include the following:

- Department of the Army (DA) Pam 73-1, *Test and Evaluation in Support of System Acquisition*
- DA Pam 73-2, *Test and Evaluation Master Plan Procedures and Guidelines*
- DA Pam 73-4, *Developmental Test and Evaluation Guidelines*
- DA Pam 73-5, *Operational Test and Evaluation Guidelines*
- DA Pam 73-7, *Software Test and Evaluation Guidelines*

2.2.3 Navy Test Policy

Naval test policy is documented in Secretary of the Navy Instruction (SECNAVINST) 5000.2b, Appendix III - Test and Evaluation. This is a high-level document that describes both developmental and operational testing. It also contains a section on software qualification testing for testing software modification releases to the fleet. There are a few sections that could be interpreted as being related to robustness testing. One of the purposes of developmental testing is to test "the effectiveness and supportability of any built-in diagnostics." This testing is related to the error-handling checks that can be done as part of

robustness testing. In addition to the 5000.2b instruction, individual Navy organizations have their own test policies.

2.3 Federal and Commercial Testing Standards

Many federal organizations have testing standards. A sample from the Federal Aviation Administration (FAA) testing guidelines states [FAA 02], "Test and evaluation programs should be structured to

- provide essential information to support decision making
- provide essential information for assessing technical and acquisition risk
- verify the attainment of technical performance specifications and objectives
- verify that systems are operationally effective and suitable for their intended use"

The FAA testing documentation covers developmental and operational testing. While developmental testing is performed mainly to ensure that the system meets contract requirements, operational testing (OT) begins to address robustness testing issues. "OT effectiveness testing evaluates the degree to which a product accomplishes its mission when used by representative personnel in the expected operational environment. This testing includes capacity and [National Airspace System] NAS loading, degraded mode operations, safety, security, and transition switchover."

The Food and Drug Administration (FDA) guidance on verification testing for software included in medical devices includes references to "robustness" types of testing for software that is considered a major area of concern in medical devices [FDA 98]. The testing should address fault, alarm, and hazard testing; error, range checking, and boundary value testing; timing analysis and testing; stress testing; communications testing; and memory utilization testing.

Commercial standards come closer to addressing robustness testing. For example, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 12207:1995 (Institute of Electrical and Electronics Engineers/Electronic Industry Alliance [IEEE/EIA] 12207.0-1996) begins to refer to what could be construed as robustness testing in the design section. Under Detailed Software Design, paragraph 5.3.6.5 states the following: "The developer shall define and document test requirements and schedule for testing software units. The test requirements should include stressing the software unit at the limits of its requirements."

Further information on references to robustness testing in commercial standards can be found in Appendix B.

3 Using Robustness Testing

3.1 General Guidelines for Robustness Testing

Robustness testing can take place at any point during system development. It can be executed as part of an overall test program or as a stand-alone test. Lyndsay points out [Lyndsay 03], "Negative testing is an open-ended activity.…Some elements of negative testing cannot be planned in any detailed way, but must be approached proactively." The categories listed below are broad areas that should be considered:

- general tests
- graphical user interface tests
- network tests
- database tests
- disk input/output (I/O) and registry tests
- memory usage tests

For any specific system, additional categories may need to be added. Keep in mind that the examples given within each category are samples. These will need to be tailored or expanded depending on the particular system being tested. Some of these tests involve modifying or corrupting data files; if these tests are going to be run, it is important to ensure that you are running in a test environment and that the testing will not affect actual system operations or files. In addition, you will need to know some basic information about the system to determine if all the tests apply. For example, the memory usage tests would not be applicable to applications written completely in Java, C#, or for a managed .Net environment because the application doesn't control the release of memory. Some of the graphical user interface tests may not be applicable in a Web-based application.

SEI personnel can view the compact disk (CD) on robustness testing for demonstrations of many of the tests listed in this section.

3.1.1 General Tests

The general tests described in this section should be performed on any system. These will provide some top-level indications of the level of detail that went into the software design and development. Lyndsay points out that, at times, just using observation, without performing a specific test, can help identify potential problems [Lyndsay 03]: "Observational

skills help the tester to spot symptoms that indicate an underlying flaw. A slow process, an extra button press, an inappropriate error message can all indicate that a system has a weakness. To the experienced tester, a weakness can be suggested by the choices available at points along a path, by a default value, by a slight change in behavior where none is necessary." These weaknesses should be considered when performing any of the tests described in this technical note.

General tests may include the tests described below.

3.1.1.1 Help Information

Check to see if there is a "help" menu. If so, check the "about" entry. A well-written "about" screen will include a real version number, compile information, and Dynamic Link-Library (DLL) information, if applicable. (For example, a Java application would not have DLL information.) Next, check to see if actual help is available. The level of detail of the help information should be consistent with the development phase. Early in the development of the product, the help section may not yet be developed, but when the software is ready to be delivered, a complete help section should be available.

3.1.1.2 Hints

Place the cursor over the icons in any of the menu bars and see if hints appear. A hint will appear in a small overlay box when the cursor is placed over the icon. The absence of hints doesn't necessarily mean the software is poorly written, but the presence of hints means that some effort was spent on developing a professional software product.

3.1.1.3 Use of Gray Out

Check to see that menu selections that are not or should not be available are properly grayed out and the icons disabled. If the icons are not grayed out, this may not indicate poor software quality. However, if the icons are grayed out, this indicates that some thought went into the development of the software.

3.1.1.4 Shortcuts

Check to see if there are shortcuts, often indicated by an underlined letter (such as ALT+F for "file") listed on the pull down menus for the application. Check at least some of the shortcuts to ensure that they work properly.

3.1.1.5 Multiple Instances

Check to see if multiple instances of an application can be run simultaneously. If running multiple instances is not allowed, ensure that it cannot happen. Check to see what happens when you try to start an application that is already running. If it is allowed, ensure that all instances work properly. The open window may need to be moved on the display to see if another instance window has opened behind it.

3.1.1.6 Recovery from Shutdown

Ensure the program recovers from an abrupt shutdown. For example, close the program using the terminate or kill command (in Windows, use the task manager to end the process), then see if the program can be restarted. Ensure that the program recovers to the previous state when restarted, if this is supposed to occur.

3.1.1.7 System Resource Usage

Check to see if there are problems with system resource usage. This includes top-level checks for memory leakage and for usage of items such as handles, threads, and graphical device interface (GDI) objects. This testing includes going to a known state of the application and using the Window's task manager or UNIX ps -aux or ps -ef to view the properties of the process or processes in question, and then noting the parameters of interest. After running the application for a period of time, return to the same base state and recheck the parameters. A large increase in any of these parameters may indicate a problem area. These top-level tests will not definitively expose a problem, but they may indicate areas that need further investigation.

3.1.1.8 Time Synchronization

If the application depends on a synchronized time source, ensure that disruption and/or corruption of the time signal is appropriately handled. You should set the time source to an abnormal time, turn the time source off, induce a jump in the time (forward and/or backward), or disconnect the time source from the application platform. Ensure the error is caught and properly handled.

3.1.2 Graphical User Interface Tests

The following tests try to expose failures in the graphical user interface (GUI). GUI failures include input-related problems, window resizing problems, and window presentation problems. While testing the GUI for the errors described below, you should also watch the display to ensure that the screen refreshes properly when changes are made. While some problems in this area won't cause fatal program errors, unexpected input values and problems with modal windows can cause fatal errors.

3.1.2.1 Invalid Input

These tests try to ensure that invalid user inputs are not accepted. In most cases, it would be extremely difficult to test every instance of user input, so select some representative inputs from different screens or different sections of the program and test some invalid inputs. It may take some effort to determine what the valid ranges are for some inputs. Both valid and invalid input information should be captured during requirements analysis and decomposition to support requirements-based verification testing. Take note of the response to invalid data: Is it simply accepted, is the data rejected without any warning message, or is a warning displayed? One condition to look for is incorrect input of time information. Some programs

may “roll over” 80 minutes to an input of 1 hour and 20 minutes. Some examples of invalid inputs to consider during testing include the following:

- Enter numbers that are out of range (too large, too small, negative).
- Enter data of the wrong type (i.e., enter alpha characters in a numeric field and vice-versa, or mixed characters and numbers).
- Enter dates that should be invalid (too early, too late, incorrect, invalid Julian dates) and incorrect date formats (i.e., incorrectly entering mm/dd/yyyy formats).
- Check for February 29th in non-leap years.
- Enter time information incorrectly (e.g., roll over of seconds and minutes, incorrect hh:mm:ss format).
- Include commas in a large number.
- Enter a large number using exponential notation.
- Enter with incorrect punctuation. (For example, enter a Social Security Number without dashes when the application is expecting dashes to be included.)
- Enter text that violates case-sensitivity rules, if applicable.
- Enter too many characters in a fixed-length field.
- Leave a value blank and just press enter, leaving the value to default (or not default as the case may be!).
- Enter special characters, such as /, \, $, %.
- Enter invalid file names. For example, file names generally cannot include special characters such as \, /, ?,), (, “, <, >, |. If the application is saving information to a file and asks for a file name, ensure it properly handles invalid names.

3.1.2.2 Modal Dialog Boxes

When a window or form is opened and it is modal, the user must explicitly close it (or provide a response so that it closes) before working in another running window or form. Most dialog boxes are modal. If a window or form is supposed to be or should be modal (usually due to internal consistency issues related to the software state, deadlock issues, or to ensure the proper user input sequence) and it is not, there is a potential for state consistency problems, unintended actions, deadlock, data loss, crash, etc. A tester will need to check that any dialog boxes that should be modal are (i.e., that you can’t perform another function or open this same window again until the first dialog box is closed). The tester should also confirm that the modal dialog box remains as the top-most window.

One example of a problem with a modal dialog box not staying on top is the Adobe Acrobat “check for updates” window. The dialog box that asks if you want to check for updates should be modal, but it is not, so if another window is opened on top of it, the portable

document format (PDF) file does not open. The user does not realize a response is needed to the “check for updates” dialog box.

To test for modality, you can try to click on another open window and/or try to open the same dialog box a second time. You may need to move the first box to ensure that another instance hasn’t opened directly behind the first box.

3.1.2.3 Resizable Windows

For this test, you will need to know which windows should be resizable and which should not be resizable when the edges are “grabbed” by the mouse cursor. For any windows that should not be resizable, ensure that they are not. For those windows that can be resized, ensure that they resize correctly. This test includes checking for scroll bars when an entire field can no longer be seen due to the resizing. Resizable windows should also be checked to ensure they can’t be closed completely during resizing. A window that can be closed down to a line should also be re-opened during this test. You should also verify that all windows minimize and restore correctly.

3.1.2.4 Combo and List Box Input

If there are combo and list boxes that should not accept user input, then ensure that data cannot be entered. If user input is acceptable, ensure that only valid inputs are allowed. If the answer is filled in as the user starts to type, this once again demonstrates the level of professionalism applied to the software development effort.

3.1.2.5 Window and Panel Splitters

If there are windows that use window splitters or panel splitters, try to move the splitter all the way to the left and then to the right. (For an example, in Windows select “Search” or “Find” from the “Start” menu, then click on the “Files and Folders” window.) Make sure the panel or sub-panel cannot be totally closed such that it cannot be reopened. Note that sometimes a panel or window will appear to be totally closed, but it can be reopened by carefully clicking (grabbing) and dragging the border. You should also check that resizing the entire window does not affect the splitters. Try moving the splitter all the way to left and making the overall window slightly smaller. Now ensure that the splitter can still be moved back to the right.

3.1.2.6 Grid GUI Interfaces

Ensure that any grid interfaces properly resize. An example of a grid interface can be seen in the “File” “Open” box of an application. The individual “Name,” “Size,” and “Type” columns are grids. This test is similar to the window and panel splitter test. The idea is to try to move grids to the extreme right and left (or up and down). If there are several grids in a specific GUI, you may need to try several combinations of adjustments to fully test the possibilities. As above, a grid may appear to close fully, but it might be reopened by clicking and dragging the border. Once again, try resizing the entire window to ensure that this does

not affect the ability to regain control of a “closed” grid. If grids can be closed completely, ensure that they can be recovered when the window or application is closed and reopened.

3.1.2.7 Multiple Instances of Open Windows

If there are windows that should not allow more than one instance to be open at one time, ensure that only one instance can be opened. Even if the application doesn’t open a new instance, check to see what happens. Does the application revert to the window that is already open, does it present an error message, or does it just not open a new window? To verify that another window has or has not opened, move the first window to the side of the display page (away from where it was when it opened).

3.1.2.8 Copy and Paste

If copy/paste is allowed, verify that it works properly. This should include a test that attempts to paste graphics into a text field. To properly test this, you may need to create a text file from which to copy and paste in a word-processing or graphics application. You should also try to copy and paste text into a numeric field and vice versa. Even if you cannot type the invalid input into a field, you may be able to paste it into that field. This test should also exercise the “Undo” option for the paste operation. You should also try copying text from the application under test into another document if users will be allowed to do this.

3.1.2.9 Drag and Drop

Verify that drag and drop can be used only where permitted. If it is permitted, try to drag invalid input into the field in question. For a more robust test, try to drag and drop objects such as text files or executables into a data field and see how the application responds.

3.1.2.10 Group Boxes and Radio Buttons

Verify that group boxes and radio buttons work correctly. Group boxes are a set of boxes that can have one or more boxes checked at the same time. When examining group boxes, look for dependencies between the state of one box and other boxes, as well as boxes being grayed out appropriately (usually when the state is mixed, i.e., some selected items have that box checked and others do not). An example of this is in Microsoft Word; under the Tools menu, select Options, and click on the Save tab. You can see here that checking the “Embed True Type Fonts” group box makes other boxes available. One other aspect of group boxes that can be tested is to ensure that if you select boxes and a cancel or default button is available, those boxes correctly reset the box selections.

Radio buttons should allow only one button to be active at any one time (for example, in the Microsoft Word “Print” screen, the selections for “All,” “Current Page,” and “Pages”). Note that on this screen, the “Selection” radio button is grayed out unless you actually have text selected. You can also check to make sure that radio buttons change correctly based on the state of the application. Check to ensure that only one radio button can be selected at any one

time, and that at least one button is always selected. If there are several buttons, you should try different combinations.

3.1.2.11 Default System Font Size

This is a slightly more severe test than most of the others, but if your system will be used by remote users who may have different system set-ups, then you may want to try this test to at least understand what would occur if the system desktop font is changed. This test is especially important if the software will be used by people with disabilities who may require the use of larger system fonts on a regular basis. It is best to change the font size when the application you are testing is closed.

The system desktop font is changed in the display properties window. (To change the font size in Windows, under the Start menu, select Settings, Control Panel, Display, Appearance. In Windows XP, you can select Extra Large Fonts; in Windows 2000 you will need to change the Scheme to Windows Standard Large or Windows Classic Large). Once the desktop font is changed to a larger font, reopen the application and check several windows, especially those where a larger font may cause problems (for example, screens that have long headings, buttons with large titles in relation to the button size, buttons that are close together, and small grids that may not display properly with a larger font size). Ensure that complete headings and button titles can be seen or that scroll bars are provided. Some programs may not use the default system font. If that is the case, ensure that this is acceptable to the user.

An even more severe test involves changing the windows font in addition to the desktop font. To change the windows font, under the Start menu, select Settings, Control Panel, Display. Then in the Settings Tab, click on the Advanced button, and change the font size to Large Fonts. You will need to restart the computer for these changes to go into effect. Restart the application and look for the effects of the font changes, especially in headings, on buttons, and in forms with constrained areas.

For non-Windows based systems, you may need to ask what alternative fonts would be available to users and determine how best to test these fonts.

Another, more advanced test involves changing the screen resolution so you can see what impact this change has on the display. For some applications, a different screen resolution can make parts of the display non-viewable.

3.1.2.12 Entry Order

Another error that can occur in the user interface is due to entering information in an unexpected order. The application often expects the user to enter information in a defined order, and if that order isn’t followed, errors could result. While testing for invalid input, you can also try entering information is an unusual order.

3.1.3 Network Tests

Consider performing at least some of the following tests on any system that connects to a network. These tests are especially important for systems where a network connection is vital to system performance.

3.1.3.1 Network Port Unavailable

Verify that the program appropriately detects, handles, and reports when a needed network port is unavailable (i.e., possibly in use by another program or in a close-wait state). Note that sometimes the absence of a resource can actually be caused by a delay in response and not a missing resource. This situation may not be testable unless the program accepts incoming network connections.

3.1.3.2 Lost Network Connection

Unplug the network cable during different phases of operation. These phases can include start-up and during any specific operations that are network dependent. Verify that lost network connections are detected, handled, and appropriately reported. Verify that the connection is properly re-established when the cable is restored, that the re-connection is appropriately reported, and that the application continues execution as appropriate.

3.1.3.3 Loss of Some Network Connections

Verify that the system is able to operate properly in a degraded mode (i.e., when one or more network connections are lost or could not be established so there is only partial connectivity). This test requires the ability to shut down other computers. For this test, you want to ensure that the program operates properly based on the connectivity that is still available. For example, you want to ensure that functions that are no longer available are properly grayed out or are handled appropriately.

3.1.3.4 Program Termination

Verify that network connections are properly terminated when the application terminates normally or abnormally. Terminate the program normally, and ensure that the network connections have been terminated. To ensure that network connections have been terminated, you can open a command prompt. (In Windows, select Run from the Start menu, and type CMD and then netstat.) Next terminate the program abnormally (in Windows, use the Task Manager) and, again, check the network connections. Note: It could take up to 90 minutes for abnormally terminated connections to reset.

3.1.3.5 Corrupt Network Data

This is a more invasive test that may not be appropriate in all situations. You can do a simple test using the telnet utility to send random character strings to a known port. A better method of testing uses external tools to input corrupt data. You would need to send malformed messages using User Datagram Protocol (UDP) and/or Transmission Control Protocol (TCP)

and make sure the application properly detects and handles these malformed messages. Appendix C contains references to tools that could be used for this test.

3.1.3.6 UDP Tests

If the application uses UDP as a communications protocol, you may want to check to ensure that the application handles lost messages and that it can detect and handle messages from unexpected hosts. You should ask the developer if the data are validated on input. For UDP transmissions, the application should check to determine where the incoming messages originate, and it should reject any messages from unexpected sources. The application also must be able to handle messages arriving out of sequence. To check this, you would need to use a tool to open a simple socket and send a message. You would then check the application to see if the message was accepted, rejected, and/or logged.

3.1.4 Database Tests

If you are testing an application that uses a shared database, you should consider using some or all of the tests in this section. Some tests may not be applicable to specific system implementations, so you may have to gather additional information on the application to decide which tests should be performed.

3.1.4.1 Database Login Failures

Ensure that the database properly handles and reports failures to login to the database. This test will require disabling the database by disabling or unplugging the server or by renaming the database and then trying to access it from the application. Sometimes the absence of a resource can actually be caused by a delay in response and not a missing resource.

3.1.4.2 Lost Database Connection

This test is closely related to network tests in the previous section. In this test, you will verify that the program attempts to reconnect to the database(s) when a connection is lost. If the database is hosted on a separate server, this test can be performed by disconnecting that server from the network. If the application is continuously connected to the database, ensure that an appropriate message is displayed when the database disconnects and another message is displayed when the connection is re-established. Ensure that the application does not hang when attempting database access while the database is down.

3.1.4.3 Database Locking/Updates

Verify that the program or system appropriately locks database records during an update or transaction. This test will require two or more users to access the database simultaneously. Another user should attempt to access a record that is being updated by the first user. Proper locking of the database during updates and transactions is vital for ensuring the integrity of the database.

The requirements for distributing database updates will vary from system to system. Ensure that the application receives updated database records when they are changed by a different user or another application per the requirements. This test requires two or more users to be accessing the database.

3.1.4.4 Corrupt Database Records

This is more severe test and will not be appropriate in all situations. This test verifies that the program can handle corrupt database records. To perform this test, a dummy database record will need to be created and then modified to produce a record that cannot be properly read by the system (for example, a record that is missing a field). Create the corrupt database record and then use the file containing that record in the application. Ensure that the corrupt record is caught and the appropriate error message is produced.

3.1.4.5 Malformed/Malicious Queries

If the user is allowed to enter ad-hoc queries from a user entry screen, a more advanced robustness test would ensure that the application could handle malformed or malicious queries. If ad-hoc queries aren't properly checked, a user could possibly access and/or change database records by entering a query in place of a variable. Even if the query is not malicious, a malformed query should not cause the application to hang or crash.

3.1.4.6 Database Transaction Errors

This test, which is closely related to the above test, is performed to see what happens when there are database transaction errors. This test verifies that data transactions are rolled backed and are not partially committed when an error is detected. This test entails disrupting the application as it attempts to save a database record. You will need to modify a database record, start to save that record, and then disrupt the save by disconnecting from the database, killing the application, etc. Restart the program and ensure that the last good version of the database is used and not the one that was being written during the disruption.

3.1.5 Disk I/O and Registry Tests

As Whittaker points out [Whittaker 03], "Many testers ignore attacks from the file system interface, assuming that files will not be a source of problems for their application. But files and the file system can cause an application to fail. The inputs from a file system are in every way, shape and form the same as inputs from a human user." These tests are more invasive than the other tests described in this document, and they may not be appropriate for all robustness testing situations. These tests ensure that files used by the application are handled properly.

3.1.5.1 Temporary Files

Verify that temporary files are deleted or reset when the application is restarted after normal and abnormal program termination. You will need to know if any temporary files are created

by the application. You could also find any temporary files by searching for files created since the application started prior to exiting the application. Ensure that temporary files are deleted when the application closes, either normally or abnormally.

3.1.5.2 Remote File System Errors

This test is done to ensure that the application can handle the situation where a remote file server is unavailable. If the software depends on a remote file server, it must be able to handle the situation when the server is not available and either shut down gracefully, or, if possible, allow remaining functionality to be used. To perform this test, the remote file server would be taken offline, and the results of accessing that server would be observed.

3.1.5.3 Missing Files

This test is done to verify that missing file or "file not found" errors are appropriately reported and handled. To test this area, you will need to rename or remove a file (or files) opened by the application. Ensure that there is an appropriate error message when the application tries to access the file and that the program fails gracefully if this does cause failure. You can also test for errors caused by incorrect file permissions.

3.1.5.4 Corrupt Files

This is an advanced test that will not be done in all situations. This test should be done if the application uses configuration files or registry files. This test will verify that corrupted data and/or configuration files (including the registry, if applicable) are detected and errors are appropriately reported and handled. You will need to go into "regedit" (registry edit) and change the path for one or two of the files. Minimize the regedit window and restart the application. Some applications will automatically correct the registry files, other programs may provide an error message, and others may simply not work.

3.1.6 Memory Tests

Memory tests beyond those described in Section 3.1.1.7 are more invasive, but they may be necessary if there is any question regarding how memory allocation is handled by the application. Extended memory testing requires running other applications that will cause a low memory condition. This testing is done to allow the tester to observe how the application handles memory utilization problems. There are also tools available that can assist with this type of testing; see Appendix C for references.

4 Applications of Robustness Testing

DoD acquisition programs can use robustness testing principles throughout the acquisition process. Anytime a software application is being demonstrated or tested, it is reasonable to at least consider adding some aspects of robustness testing. While this section primarily addresses DoD programs, it would also hold true for federal and commercial software acquisition projects.

4.1 Independent Technical Assessments (ITAs)

The SEI is often called upon to conduct ITAs. The extent of robustness testing during an ITA will depend on several factors, including the phase of software development, the purpose of the ITA, the time available, and the relationship with the organization developing and acquiring the software. You do not need to have an in-depth understanding of the internal workings of the application to execute many of these robustness tests.

There are two main ways that robustness testing can be used during an ITA. The first use of robustness testing is during a demonstration. The second use of robustness testing is to perform a top-level investigation into the quality of the software being developed. One other advantage of conducting robustness testing during an ITA is the ability to see how the developing organization reacts when problems are encountered. If an organization immediately enters the problem into their defect-tracking system and appears glad that the problem was discovered, that is a good sign. If, instead, the organization has an excuse for every problem ("the users are trained not to enter incorrect values," "there is redundancy so the database will never go down," etc.), then you should pay close attention to the attitudes of the development and test teams to ensure that there is adequate attention being paid to the quality of the application.

4.1.1 Demonstrations

Often there will be an opportunity to see a demonstration of the system during an ITA, which presents a perfect opportunity to carry out a limited robustness test. The main purpose of this testing during an ITA is simply to provide an early indication of the level of detail that went into the design, coding, and testing of the application.

An easy way to start is to ask to see the help menu (Section 3.1.1.1). Other general tests that could be easily accomplished during an ITA demonstration include those described in Sections 3.1.1.2–3.1.1.6, 3.1.2.1–3.1.2.10, and 3.1.3.2.

If the results of the robustness testing are good, that provides at least a top-level indication that the design and code have been approached professionally. However, good results from robustness testing do *not* mean that there is no need for additional questioning regarding code design and development processes and a review of the code. Especially during ITAs where time is short, robustness testing does provide additional information beyond what can be gained from interviews and code reviews alone. If there are problems encountered during the demonstration, they can help point to other areas to investigate during the interview process and code review sessions.

4.1.2 Code Quality Investigation

Some ITAs include a goal of determining the quality of the code under development. Robustness testing can be a good tool to help make this determination. Although it can't be the only tool used, it can provide some indication of how the code was developed, especially in the area of error detection, correction, and handling. The tests discussed in the demonstration section above can be augmented with the tests in Sections 3.1.3.1–3.1.3.6, 3.1.4.1–3.1.4.5, and 3.1.5.1–3.1.5.4.

4.2 Robustness Testing During Source Selection

There are two aspects of source selection that can use robustness testing:

- ensuring that the contractor considers robustness testing during system development
- testing during any source selection demonstrations

4.2.1 Provisions for Contractor Inclusion of Robustness Testing

The acquiring organization can take steps to ensure the developed product will be tested for robustness even before the source selection starts. The main goal is to ensure that contractually binding documents such as the Statement of Work (SOW) or the Integrated Master Plan (IMP) include robustness-testing principles. The Request for Proposal (RFP) can also include provisions for a software development plan to be part of the compliance documents. The acquiring organization can ask for items such as the error-handling model (the system-level approach to detecting and responding to errors) to be included in the plan. If an error-handling model is developed, the testing to ensure that it is properly implemented would be considered a subset of the robustness testing, and the acquiring organization should ask for robustness testing to be included in the testing documentation produced for the product. This testing would also be reflected in the section of the IMP dealing with test.

4.2.2 Robustness Testing During Source Selection Demonstrations

For programs using COTS software components, a demonstration is sometimes included as part of the source-selection activities. Robustness testing during source selection can also be performed on non-developmental item (NDI) software, prototype software, software developed under a previous contract phase, and/or software developed for the source selection that is considered by the developer to be production quality. If robustness testing is performed during source selection, it is important to understand the development level of the software. If the software is presented as a prototype, it may not have all the error-handling code included. If it is presented as a mature COTS product, it should pass most, if not all, of the robustness tests. Even if the demonstration software is a prototype, robustness testing during source selection can point out areas that need attention during the contract execution. Robustness tests that are appropriate during source selection include those described in Sections 3.1.1.1–3.1.1.4 and 3.1.2.1–3.1.2.10.

If robustness testing is a part of the source-selection evaluation, ensure the evaluation criteria included in the RFP mention that robustness testing will be included. The test procedures to be used must be written down and followed exactly. To avoid award protests, you must test all products exactly the same way. This will constrain the testing to what has been planned in advance, so thinking through which test types are valuable is a very important step in the process. Be sure to consider thoroughly how the results of the testing will affect the source selection. Care will be needed to plan the test and evaluation criteria such that several small, easily correctable problems do not outweigh larger issues.

Using robustness testing during source selection can help the acquiring organization gain a better understanding of the professionalism and care that has gone into producing a product. It can also provide an early indication of larger problems in critical areas such as error detection and correction.

4.3 Robustness Testing During Software Development

In most acquisitions, the acquiring organization does not get involved in the details of the contractor's software development effort. Even so, robustness-testing principles can be applied during the design and development and during unit testing. The issues raised by the various tests described above can provide a starting point for asking questions about how these areas are handled by the software under development. The idea of robustness testing is not to spring this testing on the developer at the end of the development as a "pop quiz" but to let the principles behind this type of testing guide the software development from the start.

The use of robustness testing definitely applies to the area of error handling. While attending meetings during the development phase, ask to see the error-handling model and start to ask how specific error types that are included in the robustness testing will be handled. Also, when reviewing the software development plan, if there is no mention of robustness testing, it

can be raised as a risk. If the acquiring organization has the opportunity to review development folders and unit test documentation, look for indications of robustness testing. If the contractor consistently includes robustness testing in their plans and test procedures, it is a positive sign that the contractor is concerned with the overall quality of the product.

The acquisition organization must be aware that robustness testing is not free. It will take both dollars and time to include it in the overall test program. Lyndsay suggests a three-pronged approach for estimating the cost for what he calls negative testing [Lyndsay 03]. The first type of negative testing suggested is scripted tests, which should be fairly easy to estimate. The second type is primary negative testing, which is concerned with things like failure modes, observation of failure, risk model assessment, and finding new problems. He suggests that formal techniques and checklists can help with the cost estimating and also suggests that failure mode analysis can help indicate the needed test resources. The third type of negative testing is secondary negative testing, which is testing that is found to be necessary after the start of testing. Secondary negative testing includes tests to find failures after a weakness has been found. The developer should include allocated management reserve in both cost and schedule to allow for secondary negative testing.

Often, there is a set budget for testing, and other tests may have been shortened or skipped to allow for negative testing. Time and budget must be made available to determine the relative merits of the planned testing. In every case, robustness testing should at least be considered in the test strategy.

4.4 Robustness Testing During Development Test

The acquiring organization may have slightly more involvement in the development testing of the software, but the involvement will still likely be limited. Even so, if there is no mention of robustness testing when reviewing the software test plan documentation, it can at least be raised as a risk. If the acquiring organization is invited to observe developmental testing, some of the robustness testing procedures can be requested. Robustness testing is generally performed during integration testing, but some level of robustness testing can also be accomplished during unit testing. (For example, code should be tested for invalid user inputs and any other error conditions that might arise in that module.)

The use of robustness testing can have an even greater effect in assisting a program office during system testing. The robustness testing principles described in Section 4.3 can also be used during system test and should be part of the system test plan. In addition, network connections, database connections, hardware failures, and component interactions can all be tested for robustness. There are automated test tools that can assist in identifying and preventing problems such as uncaught runtime exceptions, functional errors, memory leaks, performance problems, and security vulnerabilities. References for test tools can be found in Appendix C.

Like most software defects, if errors are caught early by robustness testing, they are easier to fix, and less retesting is required. When reviewing developmental test plans, the areas discussed in the following sections can be explored to get a better idea of the level of robustness testing planned.

4.4.1 Staff Experience

The acquiring organization can ask about the testing staff's experience with robustness testing. Much of the knowledge needed to perform beneficial robustness testing comes from coding and testing experience. In addition, there are courses available that teach robustness testing concepts. If the test team is composed entirely of young testers with no training (as is often the case), there is less likelihood that significant robustness testing will be accomplished.

4.4.2 Tools

There are tools available that can assist in the performance of robustness testing. References for test tools can be found in Appendix C. These tools include products to help load the system to assist in stress testing and tools to help check for coverage. Tools to compare "before" and "after" values in files can also be useful in robustness testing. In addition, the test team may have to develop their own tools to carry out the full range of robustness tests. These tools may include programs to use memory, test network connections, and test databases. The test team may even require special hardware to facilitate the testing. Separate servers may be needed to isolate the robustness testing from other systems, or additional hardware to simulate additional users may be needed.

4.4.3 Management Support

If robustness testing is to be successful, it must have the support of the project management teams of both the developing and acquiring organizations. Lyndsay points out [Lyndsay 03], "The idea that someone should deliberately spend their time trying to break the system in increasingly esoteric ways will seem extraordinary to many people outside the test team." If management does not let the test team know they value robustness testing, it can often be dropped when the protests of other team members are added to the pressures of budget and schedule.

4.5 Robustness Testing During Operational Testing

Within the DoD, the acquiring organization or a DoD test organization generally controls the operational test of the system. Even a commercial organization usually includes some type of acceptance testing as part of an acquisition. Robustness testing certainly can and should be implemented during operational testing. Robustness testing can be aided by the use of tools

such as fault injectors, system inspectors, or comparison tools. Although these tools are too complex to cover in detail in this report, good resources for information on software test tools can be found in Appendix C along with references for performing the more detailed robustness testing associated with operational testing of software.

4.5.1 Determining Test Cases

4.5.1.1 Error Handling

If only one area is considered during robustness testing, it should probably be error handling. While this might not be true for some specialized systems, error handling generally is a crucial area for ensuring proper system functionality. Lyndsay suggests the following [Lyndsay 03]: "Test the effectiveness and robustness of the exception handling system early—the results of later tests will depend on its reliability and accuracy." He also points out that error-handling problems may be detected at some point well after the problem occurred.

4.5.1.2 Boundary Value Analysis

Boundary value analysis is a method to help select input test cases. For each input value to be tested, a value at the boundary of acceptable input is tested, and then a value as close as possible to either side of the boundary is tested. The value on the out-of-range side is considered a "robustness" test.

4.5.1.3 Test Against Constraints

Lyndsay discusses testing the system against known constraints [Lyndsay 03]. For example, if the system constraints are that the application has to work with Internet Explorer 4.0 or later, then the robustness test might test with Explorer 3.5 or Netscape. The system doesn't necessarily have to function normally with these systems, but it should fail gracefully and not bring down the entire system.

4.5.1.4 Concurrency Testing

This robustness test checks for use of system resources. As Lyndsay points out, the tester will normally need to use simple, custom-built tools to make use of a resource before the system does [Lyndsay 03]. The system response to the busy resource should be tested. Then, the resource should be released and the tester should check that the second requestor does eventually get control of the resource. More complex tests can test for conditions such as more than two requests, queuing, timeouts, and deadlocks.

4.5.1.5 "Mis-Use" Cases

Lyndsay discusses a technique that is similar to the use of "use cases" [Lyndsay 03]. These tests help to define non-standard paths. Lyndsay's list of these types of mis-use cases for testing GUIs or browsers includes the following:

- field entry: varying order from usual sequence on form
- re-entry following rejection of input
- choosing to change existing details
- use of ‘cancel’
- use following ‘undo’
- on browsers, navigating using the ‘back’ button, bookmarks, etc.
- starting sequence halfway
- dropping out of sequence/interaction without completion/logout

There may be additional “mis-use” cases based on the specific application being tested (for example, testing a database inquiry before a connection has been made to the database).

4.5.1.6 Scalability

These tests involve stressing the system to look at issues such as maximum throughput, maximum number of users, etc. Lyndsay points out that these tests are valuable, even if fatal errors aren’t discovered [Lyndsay 03]: “A failure may not be as obvious as a system crash, and trends in response time and queue length may give an early warning. Observations made of a test system as it is driven to failure can help identify problems in a live system before they become fatal.”

4.6 Other Advantages of Robustness Testing

Robustness testing can also be used by developers to increase the quality of the software being developed. The use of robustness testing during unit test and integration test can help developers stop both coding errors and errors in the overall error-handling model that can be corrected earlier in the development.

Robustness testing can also be used during demonstrations of COTS software to get an overall idea of the product’s quality. If several COTS products are being considered, the results of the robustness testing can be added to the evaluation criteria. If a COTS product does not respond well during robustness test, a more in-depth investigation into the quality of the product should be performed before selecting the product for use in the system. If more than one COTS product will be integrated in the final system, the robustness testing should be done with as many of these products working together as possible to help identify integration problems.

5 Conclusion

Robustness testing can be a valuable addition to any software development or integration effort. This aspect of testing is often overlooked, but it can add value at almost all levels of testing. An acquirer can use simple forms of robustness testing during demonstrations to get a top-level idea of the overall quality of the software. More complete robustness testing can be used during development testing to better test the GUI and issues dealing with error detection and correction. Operational testers can use robustness testing concepts to guide more extensive testing of resource usage and stress testing.

Many organizations have realized the benefits of robustness testing. Microsoft includes negative testing experience as one of the requirements in their job listings for test engineers. A National Aeronautics and Space Administration (NASA) report found that there was too little consideration of off-nominal cases during space shuttle software testing and recommended that validation and verification performed by the developing contractors should include off-nominal cases [CETS 93]. Most books on software testing explain the concepts we have called robustness testing, even if they are called by other names.

Most of the tests described in this technical note can be run without the use of special software. Some of the more complex memory and network tests require fault injection. There are many tools that can assist with these tests. (See http://www.aptest.com /resources.html#app-func for a listing of software testing tools.)

We hope this technical note will assist personnel in evaluating software applications for their customers. We also hope this technical note will contribute to improving the practices of software testing in DoD and federal programs.

Appendix A Robustness Test Summary

Table 1 summarizes the tests described in Section 3.

Table 1: Summary of Robustness Tests

Section	General Area	Test Name	Description
3.1.1.1	General Test	Help Information	Does help exist?
3.1.1.2		Hints	Do icons have hints that appear on “mouse–over”?
3.1.1.3		Use of Gray Out	Are non-available options grayed out?
3.1.1.4		Shortcuts	Do drop-down menus provide shortcuts?
3.1.1.5		Multiple Instances	Is the attempt to launch multiple instances of the application handled appropriately?
3.1.1.6		Recovery from Shutdown	Does the application recover from an abrupt shutdown gracefully?
3.1.1.7		System Resource Usage	Does the system release resources appropriately?
3.1.1.8		Time Synchronization	Are timing issues appropriately handled?
3.1.2.1	GUI	Invalid Input	Are invalid user inputs caught and handled appropriately?
3.1.2.2		Modal Dialog Boxes	Are modal windows handled appropriately?
3.1.2.3		Resizable Windows	Do windows resize and restore properly?
3.1.2.4		Combo and List Box Input	Are combo and list boxes handled appropriately?
3.1.2.5		Window and Panel Splitters	Do window and panel splitters behave appropriately?

Table 1: Summary of Robustness Tests (cont.)

Section	General Area	Test Name	Description
3.1.2.6		Grid GUI Interfaces	Do grid interfaces resize and restore appropriately?
3.1.2.7		Multiple Instances of Open Windows	Is the attempt to open multiple instances of a window handled appropriately?
3.1.2.8		Copy and Paste	Are copy and paste prohibited where inappropriate and handled correctly where appropriate?
3.1.2.9		Drag and Drop	Can drag and drop be used only where it is permitted?
3.1.2.10		Group and Radio Boxes	Do group and radio boxes behave correctly?
3.1.2.11		Default System Font Size	Does the application interface display correctly after a change in the system default font and/or the screen resolution?
3.1.2.12		Entry Order	Does the application handle user input in an unexpected order?
3.1.3.1	Network Tests	Network Port Unavailable	Does the application respond appropriately when the needed network port is unavailable?
3.1.3.2		Lost Network Connection	Does the application respond appropriately when disconnected from the network?
3.1.3.3		Loss of Some Network Connections	Does the application work correctly when some but not all of the network connectivity is lost?
3.1.3.4		Program Termination	Are network connections properly terminated when the application terminates abnormally?
3.1.3.5		Corrupt Network Data	Can the application properly detect and handle corrupt network data?

Table 1: Summary of Robustness Tests (cont.)

Section	General Area	Test Name	Description
3.1.3.6		UDP Tests	If using UDP, can the application properly handle lost messages and messages from unexpected hosts?
3.1.4.1	Database Tests	Database Login Failures	Does the application correctly handle failures to login to the database?
3.1.4.2		Lost Database Connection	Does the application attempt to reconnect to the database if the connection is lost?
3.1.4.3		Database Locking/Updates	Does the application properly lock the database during an update or transaction? Are updated database records properly distributed?
3.1.4.4		Corrupt Database Records	Does the application properly detect and handle corrupt database records?
3.1.4.5		Malformed/Malicious Queries	If ad-hoc queries are allowed, does the application properly handle malformed or malicious queries?
3.1.4.6		Database Transaction Records	Are data transactions rolled back to the last good state if an error occurs?
3.1.5.1	Disk I/O and Registry Tests	Temporary Files	Are temporary files deleted or reset after both normal and abnormal termination?
3.1.5.2		Remote File System Errors	Does the application correctly handle the situation where a remote file server is unavailable?
3.1.5.3		Missing Files	Does the application correctly detect and handle missing files?
3.1.5.4		Corrupt Files	Does the application correctly detect and handle corrupt files?
3.1.6	Memory Tests		Does the application properly detect lack-of-memory conditions?

Appendix B Robustness Testing References in Commercial Standards

This appendix lists various commercial standards that mention robustness testing.

There are references to robustness types of testing in ISO/IEC 12207 [ISO 95]. Section 6.4.2.5, Code Verification, states the following:

> "The code shall be verified considering the criteria listed below:
>
> a) The code is traceable to design and requirements, testable, correct, and compliant with requirements and coding standards.
>
> b) The code implements proper event sequence, consistent interfaces, correct data and control flow, completeness, appropriate allocation timing and sizing budgets, and error definition, isolation, and recovery.
>
> c) Selected code can be derived from design or requirements.
>
> d) The code implements safety, security, and other critical requirements correctly as shown by suitably rigorous methods."

Both b) and d) above start to address the need for robustness testing.

In addition, Section 6.5.2, Verification, includes the following:

> "6.5.2.3 Conduct the tests in subclauses 6.5.2.1 and 6.5.2.2, including:
>
> a) Testing with stress, boundary, and singular inputs;
>
> b) Testing the software product for its ability to isolate and minimize the effect of errors; that is, graceful degradation upon failure, request for operator assistance upon stress, boundary, and singular conditions;
>
> c) Testing that representative users can successfully achieve their intended tasks using the software product.
>
> 6.5.2.4 Validate that the software product satisfies its intended use.

6.5.2.5 Test the software product as appropriate in selected areas of the target environment."

Both a) and b) above are the focus of robustness testing. The software must be appropriately stressed and checked for error-detection and error-handling capabilities.

IEEE Standard 1228-1994, *IEEE Standard for Software Safety Plans*, addresses robustness testing in paragraph e) of Section A.4, Software Safety Test Analysis [IEEE 94]:

"Software safety test analysis demonstrates that safety requirements have been correctly implemented and the software functions safely within its specified environment. Tests may include, but are not limited to, the following:

a) Computer software unit level testing that demonstrates correct execution of critical software elements.

b) Interface testing that demonstrates that critical computer software units execute together as specified.

c) Computer software configuration item testing that demonstrates the execution of one or more system components.

d) System-level testing that demonstrates the software's performance within the overall system.

e) Stress testing that demonstrates the software will not cause hazards under abnormal circumstances, such as unexpected input values or overload conditions.

f) Regression testing that demonstrates changes made to the software did not introduce conditions for new hazards."

In IEEE Standard 1012-1998, *Software Verification and Validation* [IEEE 98b],

Table 1, paragraph (5) Component V&V Test Plan Generation and Verification, and paragraph (6) Integration V&V specifically mention performance at the boundaries and performance under stress and error conditions, which are a focus of robustness testing:

"... 2) assessment of timing, sizing, and accuracy; 3) performance at boundaries and under stress and error conditions; . . ." (Note: This is for software requiring higher integrity levels.)

Appendix G of the same standard, which provides definitions, states, "Test evaluation. Evaluate the tests for requirements coverage and test completeness. Assess coverage by assessing the extent of the software exercised. Assess test completeness by determining if the

set of inputs used during test are a fair representative sample from the set of all possible inputs to the software. Assess whether test inputs include boundary condition inputs, rarely encountered inputs, and invalid inputs. For some software it may be necessary to have a set of sequential or simultaneous inputs on one or several processors to test the software adequately." The reference to boundary conditions, rarely encountered inputs, and invalid inputs is a large part of robustness testing.

The *Capability Maturity Model Integration (CMMI) - System Engineering (SE)/Software Engineering (SW), Version 1.1*, touches on the subject under the Verification Process Area [CMMI 02]. Under the Specific Practice SP1.1-1, the examples given for verification methods for software engineering include load, stress, and performance testing. In the Validation area, Generic Practice 2.3 - Provide Resources, also touches on robustness testing issues by noting that load, stress, and performance tools may be required resources.

Appendix C Resources

The following resources might be useful when dealing with robustness-testing issues.

Web Sites

http://www.aptest.com

http://www.sqatester.com/

http://www.testingeducation.org/coursenotes/

http://www.stickyminds.com/

Test Tool References

http://www.aptest.com/resources.html#app

http://www.aptest.com/resources.html#web

http://www.aptest.com/resources.html#mngt

http://testingfaqs.org/

Books and Papers

How to Break Software by James Whittaker [Whittaker 03]

A Positive View of Negative Testing by Lyndsay [Lyndsay 03]

Appendix D Acronyms

AF	Air Force
AFI	Air Force Instruction
ASP	Acquisition Support Program
CD	compact disc
CMU	Carnegie Mellon University
COTS	commercial off the shelf
DA	Department of the Army
DLL	Dynamic Link-Library
DoD	Department of Defense
DT&E	developmental test and evaluation
EIA	Electronic Industries Alliance
FAA	Federal Aviation Administration
FDA	Food and Drug Administration
GDI	graphical device interface
GUI	graphical user interface
IEEE	Institute of Electrical and Electronic Engineers
ITA	Independent Technical Assessment
M&S	modeling and simulation

NAS	National Airspace System
NDI	non-developmental item
IEC	International Electrotechnical Commission
IMP	Integrated Master Plan
ISO	International Organization for Standardization
OT	operational test
OT&E	operational test and evaluation
RFP	Request for Proposal
SECNAVINST	Secretary of the Navy Instruction
SEI	Software Engineering Institute
SOW	Statement of Work
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

References

URLs are valid as of the publication date of this document.

[ApTest 03] Aptest Software Testing Specialists, 2003. http://www.aptest.com.

[BCS 98] British Computer Society Specialist Interest Group on Software Testing (BCS SIGIST). *Software Testing. Vocabulary* (British Standard BS 7925-1). Sowerby Bridge, United Kingdom: BCS SIGIST, August 1998.

[Beizer 90] Beizer, Boris. *Software Testing Techniques.* New York, NY: van Nostrand Reinhold, 1990.

[Boehm 91] Boehm, Barry & Basili, Victor R. "Software Defect Reduction Top 10 List." *IEEE Computer 34*, 1 (January 1991): 135-137.

[CETS 93] Commission on Engineering and Technical Systems (CETS). *An Assessment of Space Shuttle Flight Software Development Process.* Washington, D.C: National Academy Press, 1993. http://books.nap.edu/books/030904880X/html/index.html.

[CMMI 02] CMMI Product Team. *CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development, V1.1* (CMU/SEI-2002-TR-004, ADA339221). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tr004.html.

[Copeland 04] Copeland, Lee. *A Practitioner's Guide to Software Test Design.* Norwood, MA: Artech House Publishers, 2004.

[FDA 98] Food and Drug Administration (FDA). *Guidance for FDA Reviewers and Industry Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices.* Rockville, MD: U.S. Department of Health and Human Services, Food and Drug Administration Center for Devices and Radiological Health, Office of Device Evaluation, May 29, 1998. http://www.fda.gov/cdrh/ode/57.html#sec3_9.

[FAA 02] Federal Aviation Administration (FAA). *Guidance: Test and Evaluation.* Rockville, MD: Food and Drug Administration, 2002. http://fast.faa.gov/test_evaluation/test_eval_toc.html.

[IEEE 94] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1228-1994: *IEEE Standard for Software Safety Plans*. New York, NY: IEEE, 1994.

[IEEE 98a] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 829: *Standard for Software Test Documentation*. New York, NY: IEEE, 1998.

[IEEE 98b] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1012-1998: *IEEE Standard for Software Verification and Validation*. New York, NY: IEEE, 1998.

[ISO 95] International Organization for Standardization/International Electrotechnical Commission (ISO/IEC). ISO/IEC 12207/1995: *Information Technology: Software Life Cycle Processes*. Genève, Switzerland: ISO/IEC, 1995.

[Lyndsay 03] Lyndsay, James. *A Positive View of Negative Testing*, London, United Kingdom: Workroom Productions Ltd, 2003. http://www.workroom-productions.com/papers/PVoNT_paper.pdf.

[Whittaker 03] Whittaker, James A. *How to Break Software*. Boston, MA: Addison Wesley, 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	April 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Robustness Testing of Software-Intensive Systems: Explanation and Guide	F19628-00-C-0003

6. AUTHOR(S)

Julie Cohen, Dan Plakosh, Kristi Keeler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TN-015

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Many Department of Defense (DoD) programs engage in what has been called “happy-path testing” (that is, testing that is only meant to show that the system meets its functional requirements). While testing to ensure that requirements are met is necessary, often tests aimed at ensuring that the system handles errors and failures appropriately are neglected. Robustness has been defined by the Food and Drug Administration as “the degree to which a software system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.” This technical note provides guidance and procedures for performing robustness testing as part of DoD or federal acquisition programs that have a software component. It includes background on the need for robustness testing and describes how robustness testing fits into DoD acquisition, including source selection issues, development issues, and developmental and operational testing issues.

14. SUBJECT TERMS	15. NUMBER OF PAGES
acquisition, Department of Defense, DoD, robustness, software testing, source selection, system testing	48

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL