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.

4.2.2 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, 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.

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:

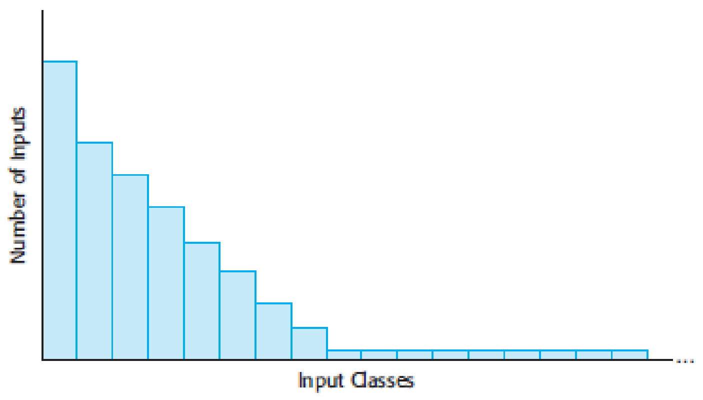

Figure 15.4 An operational profile

15.3 Security testing

The assessment of system security is increasingly important as more and more critical systems are Internet-enabled and so can be accessed by anyone with a network connection. There are daily stories of attacks on web-based systems, and viruses and worms are regularly distributed using Internet protocols.

All of this means that the verification and validation processes for web-based systems must focus on security assessment, where the ability of the system to resist different types of attack is tested. However, as Anderson explains (2001), this type of security assessment is very difficult to carry out. Consequently, systems are often deployed with security loopholes. Attackers use these to gain access to the system or to cause damage to the system or its data.

Fundamentally, there are two reasons why security testing is so difficult:

1. Security requirements, like some safety requirements, are 'shall not' requirements. That is, they specify what should not happen rather than system functionality or required behavior. It is not usually possible to define this unwanted behavior as simple constraints to be checked by the system.

   If resources are available, you can demonstrate, in principle at least, that a system meets its functional requirements. However, it is impossible to prove that a system does not do something. Irrespective of the amount of testing, security vulnerabilities may remain in a system after it has been deployed. You may, of course, generate functional requirements that are designed to guard the system against some known types of attack. However, you cannot derive requirements for unknown or unanticipated types of attack. Even in systems that have been in use for many years, an ingenious attacker can discover a new form of attack and can penetrate what was thought to be a secure system.

2. The people attacking a system are intelligent and are actively looking for vulnerabilities that they can exploit. They are willing to experiment with the system and to try things that are far outside normal activity and system use. For example, in a surname field they may enter 1,000 characters with a mixture of letters, punctuation, and numbers. Furthermore, once they find a vulnerability, they can exchange information about this and so increase the number of potential attackers.

Attackers may try to discover the assumptions made by system developers and then contradict these assumptions to see what happens. They are in a position to use and explore a system over a period of time and analyze it using software tools to discover vulnerabilities that they may be able to exploit. They may, in fact, have more time to spend on looking for vulnerabilities than system test engineers, as testers must also focus on testing the system.

For this reason, static analysis can be particularly useful as a security testing tool. A static analysis of a program can quickly guide the testing team to areas of a program that may include errors and vulnerabilities. Anomalies revealed in the static analysis can be directly fixed or can help identify tests that need to be done to reveal whether or not these anomalies actually represent a risk to the system.

To check the security of a system, you can use a combination of testing, tool-based analysis, and formal verification:

1. *Experience-based testing* In this case, the system is analyzed against types of attack that are known to the validation team. This may involve developing test cases or examining the source code of a system. For example, to check that the system is not susceptible to the well-known SQL poisoning attack, you might test the system using inputs that include SQL commands. To check that buffer overflow errors will not occur, you can examine all input buffers to see if the program is checking that assignments to buffer elements are within bounds.

   This type of validation is usually carried out in conjunction with tool-based validation, where the tool gives you information that helps focus system testing. Checklists of known security problems may be created to assist with the process. Figure 15.5 gives some examples of questions that might be used to drive experience-based testing. Checks on whether the design and programming guidelines for security (Chapter 14) have been followed might also be included in a security problem checklist.

2. *Tiger teams* This is a form of experience-based testing where it is possible to draw on experience from outside the development team to test an application system. You set up a 'tiger team' who are given the objective of breaching the system security. They simulate attacks on the system and use their ingenuity to discover new ways to compromise the system security. Tiger team members should have previous experience with security testing and finding security weaknesses in systems.

3. *Tool-based testing* For this method, various security tools such as password checkers are used to analyze the system. Password checkers detect insecure passwords such as common names or strings of consecutive letters. This

Security checklist
1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.
4. If passwords are set, does the system check that passwords are 'strong'? Strong passwords consist of mixed letters, numbers, and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.
5. Are inputs from the system's environment always checked against an input specification? Incorrect processing of badly formed inputs is a common cause of security vulnerabilities.

Figure 15.5 Examples of entries in a security checklist

approach is really an extension of experience-based validation, where experience of security flaws is embodied in the tools used. Static analysis is, of course, another type of tool-based testing.

4. *Formal verification* A system can be verified against a formal security specification. However, as in other areas, formal verification for security is not widely used.

Security testing is, inevitably, limited by the time and resources available to the test team. This means that you should normally adopt a risk-based approach to security testing and focus on what you think are the most significant risks faced by the system. If you have an analysis of the security risks to the system, these can be used to drive the testing process. As well as testing the system against the security requirements derived from these risks, the test team should also try to break the system by adopting alternative approaches that threaten the system assets.

It is very difficult for end-users of a system to verify its security. Consequently, government bodies in North America and in Europe have established sets of security evaluation criteria that can be checked by specialized evaluators (Pfleeger and Pfleeger, 2007). Software product suppliers can submit their products for evaluation and certification against these criteria. Therefore, if you have a requirement for a particular level of security, you can choose a product that has been validated to that level. In practice, however, these criteria have primarily been used in military systems and as of yet have not achieved much commercial acceptance.