CHAPTER 3

Testing Techniques

What does a tester do? In the first two chapters, our answer has been sage and learned, we hope, but also rather abstract. It's time to get more specific. Where do tests come from? What do tests look like? This chapter is about *testing techniques*, but we won't define every technique in detail. For that, you'll have to go to the main textbooks on testing. We suggest Kaner, Falk, and Nguyen (1993), Jorgensen (1995), Beizer (1990), Marick (1995), Collard (forthcoming), and Hendrickson (forthcoming). Whittaker and Jorgensen's articles (1999 and 2000) and Whittaker (2002) also provide useful ideas.

This chapter reads differently from the other chapters in the book for two reasons.

- First, the essential insight in this chapter is a structural one, a classification system that organizes the rest of the material. We placed this in the first lesson. The next five lessons list several techniques, but the primary purpose of those lists is to support the classification system. We provide this detail to make it easier for you to imagine how to apply the classification system to your work.

  This classification system synthesizes approaches that we have individually used and taught. Use this structure to decide which techniques are available and appropriate for a given problem and for generating ideas about combining techniques to attack a given problem efficiently.

  The lists of techniques sometimes contain detail beyond a quick description, but we saw that as optional. The level of detail is

intentionally uneven. We expect that you'll learn more about the details of most techniques in other books and classes.

- Second, even though this is not primarily a how-to chapter on techniques, we couldn't bring ourselves to write a chapter on testing techniques without describing at least a few techniques in enough detail that you could actually use them. Hence the *Addendum*, which describes five techniques that we find useful, in ways that have worked well for our students in professional-level seminars and university courses on software testing.

Testing combines techniques that focus on testers, coverage, potential problems, activities, and evaluation.

Our primary goal in this chapter is to present a classification system for testing techniques. We call it the *Five-fold Testing System.* Any *testing* that you do can be described in terms of five dimensions:

- *Testers. Who* does the testing. For example, user testing is focused on testing by members of your target market, people who would normally use the product.
- *Coverage. What* gets tested. For example, in function testing, you test every function.
- *Potential problems. Why* you're testing (what risk you're testing for). For example, testing for extreme value errors.
- *Activities. How* you test. For example: exploratory testing.
- *Evaluation. How to tell whether the test passed or failed.* For example, comparison to a known good result.

We also describe a few techniques in detail in this chapter and present insights about the use of a few others, but our primary goal is to explain the classification system.

All testing involves all five dimensions. A testing technique focuses your attention on one or a few dimensions, leaving the others open to your judgment. You can combine a technique that is focused on one dimension with techniques focused on the other dimensions to achieve the result you want. You might call the result of such a combination a new technique (some people do), but we think the process of thinking is more useful than adding another name to the ever-expanding list of inconsistently defined techniques in use in our field. Our classification scheme can help you make those combinations consciously and thoughtfully.

Testing tasks are often assigned on one dimension, but you do the work in all five dimensions. For example,

- Someone might ask you to do *function testing* (thoroughly test every function). This tells you what to test. You still have to decide who does the testing, what types of bugs you're looking for, how to test each function, and how to decide whether the program passed or failed.
- Someone might ask you to do *extreme-value testing* (test for error handling when you enter extreme values into a variable). This tells you what types of problems to look for. You still have to decide who will do the testing, which variables to test, how to test them, and how you'll evaluate the results.
- Someone might ask you to do *beta testing* (have external representatives of your market test the software). This tells you who will test. You still have to decide what to tell them (and how much to tell them) about, what parts of the product to look at, and what problems they should look for (and what problems they should ignore). In some beta tests, you might also tell them specifically how to recognize certain types of problems, and you might ask them to perform specific tests in specific ways. In other beta tests, you might leave activities and evaluation up to them.

Techniques don't necessarily fit on only one dimension. Nor should they; all testing involves all five dimensions, and so we should expect the richer test techniques to span several. Here's an example of what can be a multidimensional technique: If someone tells you to do "requirements-based testing," she might be talking about any combination of three ideas:

- Coverage (Test everything listed in this requirements document.)
- Potential problems (Test for any way that this requirement might not be met.)
- Evaluation (Design your tests in a way that allows you to use the requirements specification to determine whether the program passed or failed the test.)

Different testers mean different combinations of these ideas when they say, "requirements-based testing." There is no one right interpretation of this phrase.[1]

---

[1]The multiple meanings of requirements-based testing provide an example of an important general problem in software engineering. Definitions in our field are fluid. Usage varies widely across subcommunities and individuals, even when documents exist that one might expect to see used as reference standards. We'll postpone a discussion of the factors that we think lead many people to ignore the standards documents. Our point here is to note that we're not claiming to offer authoritative definitions or descriptions of the field's techniques. Some other people will use the same words to mean different things. Others probably agree with the sense of our description but would write it differently. Either position might be reasonable and defensible.

Despite the ambiguities (and, to some degree, because of them), we find this classification system useful as an idea generator.

By keeping all five dimensions in mind as you test, you might make better choices of combinations. As in beta testing, you may choose not to specify one or more of the dimensions. You might choose to not decide how results will be evaluated or how the tester will do whatever she does. Our suggestion, though, is that you make choices like that consciously, rather than adopting a technique that focuses on only one of these dimensions without realizing that the other choices still have to be made.

People-based techniques focus on who does the testing.

Here are some examples of common techniques that are distinguished by who does them.

User testing. Testing with the types of people who typically would use your product. User testing might be done at any time during development, at your site or at theirs, in carefully directed exercises or at the user's discretion. Some types of user testing, such as task analyses, look more like joint exploration (involving at least one user and at least one member of your company's testing team) than like testing by one person.

Alpha testing. In-house testing performed by the test team (and possibly other interested, friendly insiders).

Beta testing. A type of user testing that uses testers who aren't part of your organization and who are members of your product's target market. The product under test is typically very close to completion. Many companies think of any release of prerelease code to customers as beta testing; they time all beta tests to the milestone they call "beta." This is a mistake. There are actually many different types of beta tests. A *design beta*, which asks the users (especially subject matter experts) to appraise the design, should go out as soon as possible, in order to allow time for changes based on the results. A *marketing beta*, intended to reassure large customers that they should buy this product when it becomes available and install it on their large networks, should go out fairly late when the product is quite stable. In a *compatibility test beta,* the customer runs your product on a hardware and software platform that you can't easily test yourself. That must be done before it's too late for you to troubleshoot and fix compatibility problems. For any type of beta test that you manage, you should

determine its objectives before deciding how it will be scheduled and conducted.

Bug bashes. In-house testing using secretaries, programmers, marketers, and anyone who is available. A typical bug-bash lasts a half-day and is done when the software is close to being ready to release. (Note: we're listing this technique as an example, not endorsing it. Some companies have found it useful for various reasons; others have not.)

Subject-matter expert testing. Give the product to an expert on some issues addressed by the software and request feedback (bugs, criticisms, and compliments). The expert may or may not be someone you would expect to use the product—her value is her knowledge, not her representativeness of your market.

Paired testing. Two testers work together to find bugs. Typically, they share one computer and trade control of it while they test.

Eat your own dogfood. Your company uses and relies on prerelease versions of its own software, typically waiting until the software is reliable enough for real use before selling it.

Coverage-based techniques focus on what gets tested.

You could class several of these techniques differently, as problem-focused, depending on what you have in mind when you use the technique. For example, feature integration testing is coverage-oriented if you use it to check that every function behaves well when used in combination with any other function. It's problem-oriented if you have a theory of error for functions interacting together and you want to track it down. (For example, it's problem oriented if your intent is to demonstrate errors in the ways that functions pass data to each other.)

We spend some extra space on domain testing in these definitions and at the end of the chapter because the domain-related techniques are so widely used and so important in the field. You should know them.

Function testing. Test every function, one by one. Test the function thoroughly, to the extent that you can say with confidence that the function works. White box function testing is usually called unit testing and concentrates on the functions as you see them in the code. Black box function testing focuses on commands and features, things the user can do

or select. It's wise to do function testing before doing more complex tests that involve several functions. In a complex test, the first broken function will probably stop the test and block you from finding, with this test, that several other functions are also broken. If you rely on complex tests instead of testing the functions individually, you might not know until very late that one function is broken, and you might spend an enormous amount of work troubleshooting the complex test, only to discover the problem was in a simple function.

Feature or function integration testing. Test several functions together, to see how they work together.

Menu tour. Walk through all of the menus and dialogs in a GUI product, taking every available choice.

Domain testing. A domain is a (mathematical) set that includes all possible values of a variable of a function. In domain testing, you identify the functions and the variables. The variables might be input or output variables. (The mathematical distinction between input domains and output ranges is not relevant here, because the testing analysis is the same for both cases.) For each variable, you partition its set of possible values into equivalence classes and pick a small number of representatives (typically boundary cases) from each class. The assumption of the method is that if you test with a few excellent representatives of a class, you'll find most or all of the bugs that could be found by testing every member of the class. Note that in contrast to function testing, the primary element of interest is the variable rather than the function. Many variables are used by more than one function. The domain tester will analyze a variable and then, based on that analysis, run tests that involve this variable on each function with this variable as an input or an output.

Equivalence class analysis. An equivalence class is a set of values for a variable that you consider equivalent. Test cases are equivalent if you believe that (a) they all test the same thing; (b) if one of them catches a bug, the others probably will too; and (c) if one of them doesn't catch a bug, the others probably won't either. Once you've found an equivalence class, test only one or two of its members.

Boundary testing. An equivalence class is a set of values. If you can map them onto a number line, the boundary values are the smallest and largest members of the class. In boundary testing, you test these, and you also test the boundary values of nearby classes that are just smaller than the smallest member of the class you're testing and just larger than the largest member of the class you're testing. For example, consider an input field that accepts integer values between 10 and 50. The boundary values of

interest are 10 (smallest), 9 (largest integer that is too small), 50 (largest), and 51 (smallest integer that is too large).

Best representative testing. A best representative of an equivalence class is a value that is at least as likely as any other value in the class to expose an error in the software. In boundary testing, the boundary cases are almost always best representatives. But suppose that you cannot map an equivalence class onto a number line. For example, the printers that are Hewlett-Packard PCL-5 compatible are (or should be) an equivalence class because they should all work the same way. Now suppose that for a specific task, one of the printers is slightly more likely to have trouble than the others. This printer would be a best representative for that class. If it doesn't fail, we have some confidence that the other printers also wouldn't.

Input field test catalogs or matrices. For each type of input field, you can develop a fairly standard set of test cases and reuse it for similar fields in this product and later products. We give an example of this technique later in the chapter. (See the How to Create a Test Matrix for an Input Field.)

Map and test all the ways to edit a field. You can often change the value of a field in several ways. For example, you might be able to import data into the field, enter data directly into the field, have the program copy a calculated result into the field, have the program copy a recalculated result into the field, and so on. The field has constraints (restrictions on what values the field can take). Some constraints will be constant, while others will depend on the values of other fields. For example, if J and K are unsigned integers, they're constrained to the values of 0 through MaxInt. These are constant constraints. They depend on the programming language's definition of unsigned integers. However, suppose that N is also an unsigned integer, that N = J + K, and that N = 5. In this case, J = 5 − K, and J cannot possibly be bigger than 5 (the value of N). This is a variable constraint, whose range of allowable values depends on the value of N. To check that J is kept within its allowable range (5 − K), you would try changing its value using each way that you can enter data into J.

Logic testing. Variables have relationships in the program. For example, the program might have a decision rule that says that if PERSON-AGE is greater than 50 and if SMOKER is YES, then OFFER-INSURANCE must be NO. The decision rule expresses a logical relationship. Logic testing attempts to check every logical relationship in the program. *Cause-effect graphing* is a technique for designing an extensive set of logic-based tests.

State-based testing. A program moves from state to state. In a given state, some inputs are valid, and others are ignored or rejected. In response to a valid input, the program under test does something that it can do and does

not attempt something that it cannot do. In state-based testing, you walk the program through a large set of state transitions (state changes) and check the results carefully, every time.

Path testing. A path includes all of the steps that you took or all of the statements that the program passed through in order to get to your current state. Path testing involves testing many paths through the program. You cannot test all the paths through a nontrivial program. Therefore, some testers do *subpath testing*, testing many partial paths. *Basis-path testing*, for example, involves testing most or all subpaths of a certain type (the basis paths), under the assumption that if you get all of these, few tests of longer paths would be likely to find bugs that these tests missed.

Statement and branch coverage. You achieve 100 percent statement coverage if your tests execute every statement (or line of code) in the program. You achieve 100 percent statement and branch coverage if you execute every statement and every branch from one statement to another. Designing your tests to achieve a high percentage of line and branch coverage is sometimes called "Coverage-based testing." (And after you achieve that, you can quit testing or quit designing additional tests). We call this *statement-and-branch coverage* to differentiate it from all of the other types of testing that focus on some other type of coverage. Configuration coverage is an excellent example of a technique that hits the same statements many times but with potentially very different results. There are many, many other examples (Kaner 1995a). Testing that is focused on achieving high statement-and-branch coverage numbers will characteristically miss many types of bugs, such as (but not only) bugs involving missing code, incorrect handling of boundary values, timing problems, problems of compatibility with hardware and software configurations, delayed-fuse bugs like wild pointers, memory leaks or stack corruption that eventually leads to stack overflow, usability problems, and other failures to meet customer requirements. This technique is much more valuable to identify incomplete testing (what code has not yet been tested), than as a minimum standard for the amount of testing needed. Indeed, it's dangerous to allow testers to stop merely because they achieved X percent coverage (Marick 1999).

Configuration coverage. If you have to test compatibility with 100 printers, and you have tested with 10, you have achieved 10 percent printer coverage. More generally, configuration coverage measures the percentage of configuration tests that you have run (and the program has passed), compared to the total number of configuration tests that you plan to run. *Why do we call this a test technique?* Ordinarily, we would just consider this a measure of how much of a certain type of testing we had achieved. However, some testers craft a special series of tests that will make high-

volume configuration testing faster and easier. In their hands, the optimization of the effort to achieve high coverage is the test technique.

Specification-based testing. Testing focused on verifying every factual claim that is made about the product in the specification. (A factual claim is any statement that can be shown to be true or false.) This often includes every claim made in the manual, in marketing documents or advertisements, and in technical support literature sent to customers.

Requirements-based testing. Testing focused on proving that the program satisfies every requirement in a requirements document (or focused, requirement by requirement, on proving that some of the requirements have not been met.)

Combination testing. Testing two or more variables in combination with each other. We discuss this in the *Addendum on Techniques* later in this chapter. Combination testing is important, but many testers don't study enough of it. Most benefits provided by the program are based on the interaction of many variables. If you don't vary them jointly in your tests, you'll miss errors that are triggered by difficult combinations, rather than difficult individual values.

Problems-based techniques focus on why you're testing (the risks you're testing for).

Risk-based testing carries at least two major meanings.

Amland (1999) provides an excellent description of risk-based test *management*. Under this view, risk analysis is done to determine what things to test next. Testing is prioritized in terms of the probability that some feature of the program will fail and the probable cost of failure, if this feature does fail. The greater the probability of an expensive failure, the more important it is to test that feature as early and as carefully as possible.

The other meaning, which is where we're more focused, is on doing risk analyses for the purpose of finding errors. When we study a feature of a product, we ask how it can fail. That question breaks down into many additional questions, such as: What would a failure look like? Why should this feature fail—what drivers of risk are likely to have affected this feature? We describe our approach to risk-based testing in the *Addendum on Techniques.*

Both of these approaches to risk-based testing are also discussed in *James Bach on Risk-Based Testing* (1999c).

Whittaker and Jorgensen (1999 and 2000) provide excellent discussions and examples of broad classes of errors that involve constraint violations:

Input constraints. A constraint is a limit on what the program can handle. For example, if the program can only handle 32-digit numbers (or less), the programmer should provide protective routines that detect and reject an input that is outside of the 32-digit constraint. If there is no such protection, the program will fail when it attempts to process input data that it cannot process.

Output constraints. The inputs were legal, but they led to output values that the program could not handle. The program might fail when it attempts to display, print, or save an output value.

Computation constraints. The inputs and the outputs are fine, but in the course of calculating a value (that will lead to an output), the program fails. For example, multiplying two huge numbers together might yield something that is too huge for the program to cope with.

Storage (or data) constraints. Inputs, outputs, and calculations are legal, but the operations run the program out of memory or yield data files that are too enormous to process.

Whittaker (2002) provides detailed suggestions for testing against these constraints.

Here are a few additional tips for the design of risk-based tests:

- If you do risk-based testing, you must also do comparable nonrisk-based testing to test for the risk that you didn't know the risks well enough to make the right decisions.
- Test for timing issues. Surprisingly, many American-educated testers fail to consider timing issues. Some classic timing issues include race conditions and other unexpected orderings of events that happen in time.
- When you create a test, always create a test procedure that will force the program to use the test data that you have entered, allowing you to determine whether it's using that data incorrectly.

Activity-based techniques focus on how you test.

Regression testing. Regression testing involves reuse of the same tests, so you can retest (with these) after change. There are three kinds of regression testing. You do *bug fix regression* after reporting a bug and hearing later on

that it's fixed. The goal is to prove that the fix is no good. The goal of *old bugs regression* is to prove that a change to the software has caused an old bug fix to become unfixed. *Side-effect regression*, also called *stability regression*, involves retesting of substantial parts of the product. The goal is to prove that the change has caused something that used to work to now be broken.

Scripted testing. Manual testing, typically done by a junior tester who follows a step-by-step procedure written by a more senior tester.

Smoke testing. This type of side-effect regression testing is done with the goal of proving that a new build is not worth testing. Smoke tests are often automated and standardized from one build to the next. They test things you expect to work, and if they don't, you'll suspect that the program was built with the wrong file or that something basic is broken.

Exploratory testing. We expect the tester to learn, throughout the project, about the product, its market, its risks, and the ways in which it has failed previous tests. New tests are constantly created and used. They're more powerful than older tests because they're based on the tester's continuously increasing knowledge.

Guerilla testing. A fast and vicious attack on the program. A form of exploratory testing that is usually time-boxed and done by an experienced exploratory tester. For example, a senior tester might spend a day testing an area that will otherwise be ignored. She tries out her most powerful attacks. If she finds significant problems, the area will be rebudgeted, and the overall test plan might be affected. If she finds no significant problems, the area will hereinafter be ignored or only lightly tested.

Scenario testing. A scenario test (as we use the term) normally involves four attributes. (1) The test must be realistic. It should reflect something that customers would actually do. (2) The test should be complex, involving several features, in a way that should be challenging to the program. (3) It should be easy and quick to tell whether the program passed or failed the test. (4) A stakeholder is likely to argue vigorously that the program should be fixed if it fails this test. A test with these four attributes will be persuasive and will probably yield bug fixes if it fails the program. However, you might have to spend days developing an excellent scenario test.

Scenario testing. Tests derived from use cases are also called scenario tests (Jacobson 1992, Collard 1999) or *use case flow tests*. (Many people would classify these as coverage-based tests, focusing on coverage of the important use cases.)

Installation testing. Install the software in the various ways and on the various types of systems that it can be installed. Check which files are added or changed on disk. Does the installed software work? What happens when you uninstall?

Load testing. The program or system under test is attacked, by being run on a system that is facing many demands for resources. Under a high enough load, the system will probably fail, but the pattern of events leading to the failure will point to vulnerabilities in the software or system under test that might be exploited under more normal use of the software under test. Asbock (2000) is an excellent introduction to load testing.

Long sequence testing. Testing is done overnight or for days or weeks. The goal is to discover errors that short sequence tests will miss. Examples of the errors that are often found this way are wild pointers, memory leaks, stack overflows, and bad interactions among more than two features. (This is sometimes called duration testing, reliability testing, or endurance testing.)

Performance testing. These tests are usually run to determine how quickly the program runs, in order to decide whether optimization is needed. But the tests can expose many other bugs. A significant change in performance from a previous release can indicate the effect of a coding error. For example, if you test how long a simple function test takes to run today and then run the same test on the same machine tomorrow, you'll probably check with the programmer or write a bug report if the test runs more than three times faster or slower. Either case is suspicious because something fundamental about the program has been changed.[2]

Evaluation-based techniques focus on how to tell whether the test passed or failed.

The evaluation techniques describe methods for determining whether the program passed or failed the test. They don't specify how the testing should be done or how the data should be collected. They tell you that, if you can collect certain data, you can evaluate it.

Self-verifying data. The data files you use in testing carry information that lets you determine whether the output data is corrupt.

Comparison with saved results. Regression testing (typically, but not always automated) in which pass or fail is determined by comparing the results you got today with the results from last week. If the result was correct last week, and it's different now, the difference might reflect a new defect.

[2]Sam Guckenheimer noted to us, "A performance difference might also reflect a change in a third-party component or configuration. For example, changes in the JVM with different Sun releases of the JDK have had remarkably different performance characteristics. Since this is a customer-updateable component, performance testing can yield surprising results even when your code hasn't changed at all!"

Comparison with a specification or other authoritative document. A mismatch with the specification is (probably) an error.

Heuristic consistency. Consistency is an important criterion for evaluating a program. Inconsistency may be a reason to report a bug, or it may reflect intentional design variation. We work with seven main consistencies:

1. *Consistent with history.* Present function behavior is consistent with past behavior.
2. *Consistent with our image.* Function behavior is consistent with an image the organization wants to project.
3. *Consistent with comparable products.* Function behavior is consistent with that of similar functions in comparable products.
4. *Consistent with claims.* Function behavior is consistent with what people say it's supposed to be.
5. *Consistent with user's expectations.* Function behavior is consistent with what we think users want.
6. *Consistent within product.* Function behavior is consistent with behavior of comparable functions or functional patterns within the product.
7. *Consistent with purpose.* Function behavior is consistent with apparent purpose.

Oracle-based testing. An oracle is an evaluation tool that will tell you whether the program has passed or failed a test. In high-volume automated testing, the oracle is probably another program that generates results or checks the software under test's results. The oracle is generally more trusted than the software under test, so a concern flagged by the oracle is worth spending time and effort to check.

The classification of a technique depends on how you think about it.

You might be puzzled about why we placed particular techniques where we did. If so, good for you: Your brain is turned on. Remember, all testing involves all five aspects of the Five-Fold System. We've listed techniques by category simply to give you a flavor of how different techniques emphasize some ways of thinking over others. Your taste may vary. For example, one reader argued with us that load testing should be classified as a problem-focused (or risk-focused test) rather than as an activity-focused test. Our answer is that you can think of it either way.

Let's look at this from a problem-oriented perspective:

- You can think of load testing in terms of the effect of denial-of-service attacks. An attacker could attempt to deny service by creating too many connections or users or by using too much memory (have every user issue a memory-intensive command at the same time) or by using tasks that eat too much processing capacity. You could do different load tests for each of these types of risk.

Now consider an activity perspective:

- Use a tool to track the patterns of activities of your customers. Which commands do customers use most often? What tasks are customers most likely to attempt? What percentage of customers do which activities? When you have a model of the usage patterns at your site, get a load test tool and program it with scenarios that look like each of the types of use. Have that tool randomly select among scenarios—in effect, create different sessions that represent different types of users. Keep adding sessions and watch how performance and reliability of the system degrades with increased load. Make changes to the software as appropriate.

When you think in terms of the risk, you think of a weakness that the program might have and ask how to design a test series that would expose that kind of weakness. When you know what type of test you want to run, think about the way that you'll run that test. If you were testing a telephone system, you might need a tool, or you might be as well off with 10 friends who make a bunch of phone calls. The criterion for the test design is that it must have power—the ability to detect the fault that you're looking for.

In contrast, when you think in terms of activities, you're asking how to do load testing. What tools? What will the tools do? And so on. The expectation is that if you use the tools competently, model customers accurately, and do the other activities associated with good load testing, then you'll probably find the types of bugs that load testing is likely to reveal.

Either classification is accurate, but the classification itself only helps you so much. However you classify a technique like load testing, when it comes time to test, you'll still have the same five dimensions of decision:

- *Who will do the testing?*
- *What aspects of the program are you testing?*
- *What types of problems are you looking for?*
- *What tasks, specifically, will you do?*
- *How will you tell whether a test passed or failed?*

Addendum to Techniques

Here are more detailed descriptions of a few of the key testing techniques that we've found particularly useful:

- How to create a test matrix for an input field.
- How to create a test matrix for repeating issues.
- How to create a traceability matrix for specification-based testing.
- How to do combination testing using the all-pairs technique.
- How to analyze the risks associated with some item or aspect of the program.

How to Create a Test Matrix for an Input Field

Start by asking, "What are the interesting input tests for a simple integer field?" Here are some of the tests we think of as routine for a field of this kind:

- Nothing
- Empty field (clear the default value)
- Outside of upper bound (UB) number of digits or characters
- 0
- Valid value
- At lower bound (LB) of value − 1
- At lower bound (LB) of value
- At upper bound (UB) of value
- At upper bound (UB) of value + 1
- Far below the LB of value
- Far above the UB of value
- At LB number of digits or characters
- At LB − 1 number of digits or characters
- At UB number of digits or characters
- At UB + 1 number of digits or characters
- Far more than UB number of digits or characters
- Negative
- Nondigits, especially / (ASCII character 47) and : (ASCII character 58)

- Wrong data type (*e. g.*, decimal into integer)
- Expressions
- Leading space
- Many leading spaces
- Leading zero
- Many leading zeros
- Leading + sign
- Many leading + signs
- Nonprinting character (*e. g.*, Ctrl+char)
- Operating system filename reserved characters (*e. g.*, "\ * . :")
- Language reserved characters
- Upper ASCII (128–254) (a.k.a. ANSI) characters
- ASCII 255 (often interpreted as end of file)
- Uppercase characters
- Lowercase characters
- Modifiers (*e. g.*, Ctrl, Alt, Shift-Ctrl, and so on)
- Function key (F2, F3, F4, and so on)
- Enter nothing but wait for a long time before pressing the Enter or Tab key, clicking OK, or doing something equivalent that takes you out of the field. Is there a time-out? What is the effect?
- Enter one digit but wait for a long time before entering another digit or digits and then press the Enter key. How long do you have to wait before the system times you out, if it does? What happens to the data you entered? What happens to other data you previously entered?
- Enter digits and edit them using the backspace key, and delete them, and use arrow keys (or the mouse) to move you into the digits you've already entered so that you can insert or overtype new digits.
- Enter digits while the system is reacting to interrupts of different kinds (such as printer activity, clock events, mouse movement and clicks, files going to disk, and so on).
- Enter a digit, shift focus to another application, return to this application. Where is the focus?

A list like this is often called a *catalog of tests*. (Marick 1995 defines the term and gives examples.) We find it useful to put the list into a matrix form, as in Table 3.1.

Table 3.1 Numeric Input Field Test Matrix

Numeric Input Field	Nothing	Empty (clear default)	0	LB-1	LB	UB	UB+1	Far below LB	Far above UB	UB number of chars	UB +1 chars	Far beyond UB chars	Negative	Non-digit (/ ASCII 47)	Non-digit (: ASCII 58)	wrong data type	expressions	Leading spaces	Non-printing char	O/S file name	Upper ASCII	Upper case	lower case	Modifiers (Ctrl, Alt, etc.)	Function keys

Across the top of the matrix are the tests that you'll use time and time again. Down the side are the fields that you would test. For example, in the typical Print dialog, one of the fields is Number of Copies. The range of valid values for Number of Copies is typically 1 to 99 or 1 to 255 (depends on the printer). On the form, you might write *Print: Number of Copies* on one row, then run some or all of the tests on that field, and then fill in the results accordingly. (We like to use green and pink highlighters to fill in cells that yielded passing and failing test results.)

The matrix provides a structure for easy delegation. When a new feature is added or a feature is changed late in the project, you can assign several of these standard matrices to a tester who is relatively new to the project (but experienced in testing). Her task is to check for basic functioning of the new feature or continued good functioning of older features that you expect to be impacted by the one that was changed.

The integer input field is just one example. You might find it valuable to create charts like this for rational numbers (of various precisions), character fields (of various widths), filenames, file locations, dates, and so on. If you run into one type of input field program after program or time after time in the program that you're testing, it's worth spending the time to create a reusable matrix.

How to Create a Test Matrix for Repeating Issues

The text matrix for an input field is just one example of a broad class of useful matrices you can create. Input fields aren't the only candidates for standardization. Whenever a situation recurs frequently, across and within programs, you have a basis for spending time and effort to create a testing catalog. Given a catalog, you can always format it as a matrix.

Here's an example that doesn't involve input variables.

In this case, the catalog lists the ways the program could be unsuccessful in an attempt to write a file to disk. In several situations the program would attempt to write a file, such as:

- Saving a new file.
- Overwriting a file with the same name.
- Appending to a file.
- Replacing a file that you're editing, with a new version, same name.
- Exporting to a new file format.

- Printing to disk.
- Logging messages or errors to disk.
- Saving a temporary file. (Many programs do this as part of their routine, so you might not think of it during user interface testing. However, if the disk is full, the program can still fail.)

Each of these situations will have its own row in the matrix. Similarly, if the software under test enables you to export to different formats, the test matrix will have one format per row.

The columns indicate the tests that you perform. For example, try saving a file to a full disk. Try saving a file to an almost full disk. Try saving a file to a drive that gets disconnected, and so on.

Here's a catalog of some of the interesting test cases for unsuccessful attempts to save a file:

- Save to a full local disk.
- Save to an almost full local disk.
- Save to a write-protected local disk.
- Save to a full disk on the local network.
- Save to an almost full disk on the local network.
- Save to a write-protected disk on the local network.
- Save to a full disk on a remote network.
- Save to an almost full disk on a remote network.
- Save to a write-protected disk on a remote network.
- Save to a file, directory, or disk that you don't have write privileges to.
- Save to a damaged (I/O error) local disk, local network disk, or remote disk.
- Save to an unformatted local disk, local network disk, or remote disk.
- Remove local disk, local network disk, or remote disk from drive after opening file.
- Timeout waiting for local disk, local network disk, or remote disk to come back online.
- Create a keyboard and mouse I/O during a save to a local disk, local network disk, or remote disk.
- Generate some other interrupt during a save to a local drive, local network disk, or remote disk.

- Power out (of local computer) during a save to a local drive, local network disk, or remote disk.
- Power out (of drive or computer connected to drive) during a save to a local drive, local network disk, or remote disk.

To create a catalog like this, we suggest you have two brainstorming sessions with colleagues. In the first session, try to think of anything that would be a test you would routinely run on the object (like input field) or task (like saving a file) under test. Budget an hour, fill lots of flip charts, and then send your colleagues away for a day while you organize the material from the brainstorms.

To organize the material, create a new set of flipchart pages. Write a theme heading on each, like "disk capacity" and "interrupted while writing." Under the heading, copy all of the items from the flipcharts that fit into that theme. Eventually, all items will be on one of the theme flipcharts or discarded. (Feel free to discard dumb ideas.)

Next day, brainstorm using the theme charts. People will add more items to "disk capacity" and to "interrupted while writing" and so on. Have a few spare charts ready for new themes. It's not uncommon to double the length of the list in the second meeting.

After the second meeting, sort tests into essential ones, which go onto the main test matrix; the infrequently used ones, which go to a secondary list that you might distribute with the main matrix; and the discards.

Nguyen (2000) provides additional examples of test matrices.

How to Create a Traceability Matrix for Specification-Based Testing

A traceability matrix enables you to trace every test case forward to an item (items) in the specification and to trace back from every specification item to all of the test cases that test it. Table 3.2 shows an example.

Each column contains a different specification item. A spec item might refer to a function, a variable, a value (*e. g.,* a boundary case) of a variable, a promised benefit, an allegedly compatible device, or any other promise or statement that can be proved true or false.

Each row is a test case.

Each cell shows which test case tests which items.

Table 3.2 Specification Traceability Matrix

	SPEC ITEM 1	SPEC ITEM 2	SPEC ITEM 3	SPEC ITEM 4	SPEC ITEM 5	SPEC ITEM 6
Test Case 1	X		X			X
Test Case 2	X	X		X		X
Test Case 3			X	X		X
Test Case 4			X	X		X
Test Case 5	X				X	X
Test Case 6		X				X
TOTALS	3	2	3	3	1	6

If a feature changes, you can see quickly which tests must be reanalyzed and probably rewritten. In general, you can trace back from a given item of interest to the tests that cover it.

This matrix isn't a perfect test document. It doesn't specify the tests; it merely maps the test cases to the specification items. You can't see from this matrix whether the test is a powerful one or a weak one, or whether it does something interesting with the feature (or other spec item) or something that no one would care about. You also can't see the testing that is being done of features that haven't been specified or of adjustments you've made in the testing to deal with specs that were incorrect. Despite these problems, charts like this can be useful for helping you understand:

- That one feature or item is almost never tested, while another is tested extremely often.
- That a change to one item (such as Spec Item 6 in Table 3.2) will cause revisions to a huge number of tests in the system. (This is a key issue in contract-driven development, because the client is going to pay a lot of testing money if they get the change they're apparently asking for, and they should be warned of that before the change is made.)

Traceability matrices are useful in more cases than just specification-driven testing. Any time you have a list of things you want tested (specification items, features, use cases, network cards, whatever), you can run that list across the top of the matrix, run the list of test cases down the rows, and then check which test cases test what. You'll almost certainly find holes in your testing this way. If your tests are automated, you may be able to generate a traceability matrix automatically.

How to Do Combination Testing Using the All-Pairs Technique

Combination testing involves testing several variables together. The first critical problem of combination testing is the number of test cases. Imagine testing three variables together, when each variable has 100 possible values. The number of possible tests of Variable 1 with Variable 2 with Variable 3 is $100 \times 100 \times 100 = 1{,}000{,}000$ test cases. Reducing the number of tests is a critical priority.

Start with Domain Partitioning

The first step is to reduce the number of values that will be tested in each variable. The most common approach involves domain testing. Partition the values of Variable 1 into subdomains and choose best representatives of the subdomains. Perhaps you can bring the number of tests of Variable 1 down to five this way. If you can do the same for Variable 2 and Variable 3, you now only have $5 \times 5 \times 5$ tests (125). This is still too many for practical purposes, but it's a lot less than a million.

The best discussions in print of partitioning are in Ostrand and Balcer (1988) and Jorgensen (1995). Jorgensen provides good examples of partitioning and of testing several partitioned variables in combination. We present a different approach to combinations from his, which we have found useful.

Achieving All Singles

The simplest set of combination tests would ensure that you cover every value of interest (every one of the five that we'll test) of every variable. This is called *all singles* (in contrast to all pairs and all triples) because you're making sure that you hit every single value of every variable. You can achieve this as follows:

1. Let V1, V2, and V3 stand for the three variables.
2. Let A, B, C, D, and E be the five values of interest in variable V1. In particular, suppose that V1 is the operating system; A is Windows 2000; B is Windows 95; C is Windows 98 original; D is Windows 98 with the first service pack; and E is Windows ME.
3. Let I, J, K, L, and M be the five values of interest in variable V2. In particular, suppose that V2 is the browser; I is Netscape 4.73; J is Netscape 6; K is Explorer 5.5; L is Explorer 5.0; and M is Opera 5.12 for Windows.

4. Let V, W, X, Y, and Z be the five values of interest in variable V3. These refer to five different disk drive options on the system.

To test all combinations of these variables' values, we would have $5 \times 5 \times 5 = 125$ tests.

Table 3.3 is a combination test table that achieves "complete testing," when the criterion of completeness is that every value of every variable must appear in at least one test.

This approach is often used in configuration testing to reduce the number of configurations under test to a manageable number.

A serious problem with the approach is that it misses predictably important configurations. For example, a lot of people might be using Explorer 5.5 with Windows ME, but that test isn't listed here. Instead, the table shows Opera 5.12 with Windows ME.

A common solution to this problem is to specify additional test cases that include key pairs of variables (such as Explorer 5.5 with Windows ME) or more key combinations of more than two variables (such as Explorer 5.5, Windows ME, HP 4050N printer, 256M RAM, and a 21-inch color monitor being driven at 1600 × 1200 resolution.) Marketing or technical support staff might specify these, naming perhaps 10 or 20 additional key configurations, for a total of 15 or 25 tests.

Achieving All Pairs

In the *all pairs* approach (Cohen *et al.*, 1996 and 1997), the set of test cases includes all of the pairs of values of every variable. So, E (Windows ME) isn't just paired with M (Opera). It's also paired with I, J, K, and L. Similarly, E is paired with every value of V3.

Table 3.3 All Singles—All Values are Represented at Least Once

	VARIABLE 1	VARIABLE 2	VARIABLE 3
Test Case 1	A *(Win 2K)*	I *(Netscape 4.73)*	V *(Disk option 1)*
Test Case 2	B *(Win 95)*	J *(Netscape 6)*	W *(Disk option 2)*
Test Case 3	C *(Win 98)*	K *(IE 5.5)*	X *(Disk option 3)*
Test Case 4	D *(Win 98 SP1)*	L *(IE 5.0)*	Y *(Disk option 4)*
Test Case 5	E *(Win ME)*	M *(Opera 5.12)*	Z *(Disk option 5)*

Table 3.4 All Pairs—All Pairs of Values are Represented at Least Once (25 Instead of 125 Tests)

	VARIABLE 1	VARIABLE 2	VARIABLE 3
Test Case 1	A	I	V
Test Case 2	A	J	W
Test Case 3	A	K	X
Test Case 4	A	L	Y
Test Case 5	A	M	Z
Test Case 6	B	I	W
Test Case 7	B	J	Z
Test Case 8	B	K	Y
Test Case 9	B	L	V
Test Case 10	B	M	X
Test Case 11	C	I	X
Test Case 12	C	J	Y
Test Case 13	C	K	Z
Test Case 14	C	L	W
Test Case 15	C	M	V
Test Case 16	D	I	Y
Test Case 17	D	J	X
Test Case 18	D	K	V
Test Case 19	D	L	Z
Test Case 20	D	M	W
Test Case 21	E	I	Z
Test Case 22	E	J	V
Test Case 23	E	K	W
Test Case 24	E	L	X
Test Case 25	E	M	Y

Table 3.4 illustrates a set of combinations that will meet the all-pairs criterion. Every value of every variable is paired with every value of every other variable in at least one test case. This is a much more thorough standard than all singles, but it still reduces the number of test cases from 125 (all combinations) to 25, a big savings.

To show how to create an all-pairs test set, we'll work through a simpler example, step by step.

A Step-By-Step Example

Imagine a program with three variables: V1 has three possible values; V2 has two possible values; and V3 has two possible values. If V1, V2, and V3 are independent, the number of possible combinations is 12 ($3 \times 2 \times 2$).

To build the all-pairs table, start this way:

1. Label the columns with the variable names, listing variables in descending order (of number of possible values).
2. If the variable in Column 1 has V1 possible values and the variable in Column 2 has V2 possible values, there will be at least V1 $\times$ V2 rows (draw the table this way but leave a blank row or two between repetition groups in Column 1).
3. Fill in the table, one column at a time. The first column repeats each of its elements V2 times, skips a line, and then starts the repetition of the next element. For example, if variable 1's possible values are A, B, and C and V2 is two, Column 1 would contain A, A, blank row, B, B, blank row, C, C, blank row. Skip the blank row because it is hard to know how many tests (how many rows) will be needed. Leave room for extras.
4. In the second column, list all the values of the variable, skip the line, list the values, and so forth. For example, if Variable 2's possible values are X and Y, the table looks like Table 3.5 so far.
5. Add the third column (the third variable).

   Each section of the third column (think of the two AA rows as defining a section, BB as defining another, and so on) will have to contain every value of Variable 3. Order the values so that the variables also make all pairs with Variable 2.

Table 3.5 First Step in Creating the All-Pairs Matrix

VARIABLE 1	VARIABLE 2	VARIABLE 3
A	X	
A	Y	
B	X	
B	Y	
C	X	
C	Y	

Suppose that Variable 3 can have values 0 or 1. The third section can be filled in either way, and you might highlight your choice on the matrix so that you can reverse it later if you have to. The decision (say 1,0) is arbitrary. See Table 3.6.

Now that we've solved the three-column exercise, try adding more variables. Each of them will have two values.

To add a variable with more than two values, you have to start over, because the order of variables in the table must be from the one with the largest number of values to the next largest number and on down so that the last column has the variable with the fewest values. (You *could* do it differently, but our experience is that you'll make so many mistakes that you would be unwise to try to do it differently.)

The fourth column will go in easily. Start by making sure you hit all pairs of values of Column 4 and Column 2 (this can be done in the AA and BB blocks), then make sure you get all pairs of Column 4 and Column 3. See Table 3.7.

Watch this first attempt on Column 5 (see Table 3.8). It achieves all pairs of GH with Columns 1, 2, and 3 but misses it for Column 4.

The most recent arbitrary choice was HG in the BB section. (After the order of H then G was determined for the BB section, HG is the necessary order for the third in order to pair H with a 1 in the third column.)

To recover from guessing incorrectly that HG was a good order for the second section, erase it and try again:

Table 3.6 Second Step in Creating the All-Pairs Matrix

VARIABLE 1	VARIABLE 2	VARIABLE 3
A	X	1
A	Y	0
B	X	0
B	Y	1
C	X	1
C	Y	0

Table 3.7 Adding a Fourth Variable to the All-Pairs Matrix

VARIABLE 1	VARIABLE 2	VARIABLE 3	VARIABLE 4
A	X	1	E
A	Y	0	F
B	X	0	F
B	Y	1	E
C	X	1	F
C	Y	0	E

Table 3.8 Adding a Fifth Variable to the All-Pairs Matrix. (This one doesn't work, but it illustrates how to make a guess and then recover if you guess incorrectly.)

VARIABLE 1	VARIABLE 2	VARIABLE 3	VARIABLE 4	VARIABLE 5
A	X	1	E	G
A	Y	0	F	H
B	X	0	F	H
B	Y	1	E	G
C	X	1	F	H
C	Y	0	E	G

1. Flip the most recent arbitrary choice (Column 5, Section BB, from HG to GH).
2. Erase section CC because the choice of HG there was based on the preceding section being HG, and we just erased that.
3. Refill section CC by checking for missing pairs. GH, GH would give us two XG, XG pairs, so flip to HG for the third section. This yields a Column 2X with a Column 5H and a Column 2Y with a Column 5G, as needed to obtain all pairs. (See Table 3.9.)

If you try to add yet another variable, it won't fit in the six pairs. Try it with the IJs (the values of Variable 6) in any order, and it just won't work. (See Table 3.10.)

Table 3.9 Successfully Adding a Fifth Variable to the All-Pairs Matrix

VARIABLE 1	VARIABLE 2	VARIABLE 3	VARIABLE 4	VARIABLE 5
A	X	1	E	G
A	Y	0	F	H
B	X	0	F	G
B	Y	1	E	H
C	X	1	F	H
C	Y	0	E	G

However, this is easy to fix. We just need two more test cases. See Table 3.11. If you consider the second table, what is needed is a test that pairs a G with a J and another test that pairs an H with an I. The values of any of the other variables are irrelevant (as far as achieving all pairs), so fill them with anything you want. If you're going to keep adding variables, you might leave them blank, and decide later (as you try to accommodate Variable 7 and Variable 8 into the same eight test cases) what values would be convenient in those rows.

If we tried to test all of the combinations of these variables, there would be $3 \times 2 \times 2 \times 2 \times 2 \times 2 = 96$ tests. We've reduced our set of tests, using all-pairs, from 96 to 8, a substantial savings.

There are risks if you *only* use the all-pairs cases. As with all-singles, you might know of a specific combination that is widely used or likely to be troublesome. The best thing to do is add this case to the table. You've cut back from 96 to 8 tests. It's sensible to expand the set out to 10 or 15 tests, to cover the important special cases. For another worked example, see (Cohen *et al.* 1997).

How to Analyze the Risks Associated with Some Item or Aspect of the Program

Suppose you're testing some feature of the product. (You could just as well be testing a variable. This is not restricted to features.)

The feature might have a problem. That is, it might fail to live up to an important measure of the quality of the product.

To determine whether the feature has a problem, consider the problem drivers, the things that make a feature more likely to go wrong.

Table 3.10 These Six Variables Do Not Fit into the Six Tests in the All-Pairs Matrix

VAR 1	VAR 2	VAR 3	VAR 4	VAR 5	VAR 6
A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	J
B	Y	1	E	H	I
C	X	1	F	H	J
C	Y	0	E	G	I

VAR 1	VAR 2	VAR 3	VAR 4	VAR 5	VAR 6
A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	I
B	Y	1	E	H	J
C	X	1	F	H	J
C	Y	0	E	G	I

Table 3.11 All Pairs with Six Variables in Eight Test Cases

VAR 1	VAR 2	VAR 3	VAR 4	VAR 5	VAR 6
A	X	1	E	G	I
A	Y	0	F	H	J
				G	J
B	X	0	F	G	I
B	Y	1	E	H	J
				H	I
C	X	1	F	H	J
C	Y	0	E	G	I

Quality Attributes

If a feature lacks or violates any of these attributes, it's probably due for a bug report:

- Accessibility
- Capability
- Compatibility
- Concurrency
- Conformance to standards
- Efficiency
- Installability and uninstallability
- Localizability
- Maintainability
- Performance
- Portability
- Recoverability
- Reliability
- Scalability
- Security
- Supportability
- Testability
- Usability

To determine whether a feature is defective, ask yourself how you would prove that it lacks or violates one of these attributes.

For example, consider *usability*. How could you prove that the feature under test is unusable? What would unusability look like? What traditional usability tests could you apply to study this feature? Ask questions like these (and run appropriate tests).

These tests are constrained by your imagination, but many of the ideas that could enter your imagination might come from the problem drivers list.

Problem Drivers

Here are some factors that suggest a likelihood of errors to us. You can treat each of these as a small or large (your judgment) warning flag and design tests to determine whether the program actually has the vulnerability that these factors suggest.

New things. Newer features may fail.

New technology. New concepts lead to new mistakes.

New market(s). A different customer base will see and use the product differently.

Learning curve. Mistakes are made because of ignorance.

Changed things. Changes may break old code.

Late change. Rushed decisions, rushed or demoralized staff lead to mistakes.

Rushed work. Some tasks or projects are chronically underfunded, and all aspects of work quality suffer.

Poor design or unmaintainable implementation. Some internal design decisions make the code so hard to maintain that fixes consistently cause new problems.

Tired programmers. Long overtime over several weeks or months yields inefficiencies and errors.

Other staff issues. Alcohol problems, health problems, a death in the family . . . . Two programmers who won't talk to each other (neither will their code) . . . .

Just slipping it in. A programmer's pet (but unscheduled) feature may interact badly with other code.

N.I.H. (Not Invented Here) External components can cause problems.

N.I.B. (Not In Budget) Unbudgeted tasks may be done shoddily.

Ambiguity. Ambiguous descriptions (in specs or other docs) can lead to incorrect or conflicting implementations.

Conflicting requirements. Ambiguity often hides a conflict; the result of which is the loss of value for some person.

Unknown requirements. Requirements surface throughout development. Failure to meet a legitimate requirement is a failure of quality for that stakeholder.

Evolving requirements. People realize what they want as the product develops. Adhering to a start-of-the-project requirements list may meet the contract but fail the product. (Check out www.agilealliance.org.)

Complexity. Complex code may be buggy.

Bugginess. Features with many known bugs may also have many unknown bugs.

Dependencies. Failures may trigger other failures.

Untestability. Risk of slow, inefficient testing.

Little unit testing. Programmers find and fix most of their own bugs. Shortcutting here is a risk.

Little system testing so far. Untested software may fail.

Previous reliance on narrow testing strategies. For example, regression and function tests, can yield a backlog of errors surviving across versions.

Weak testing tools. If tools don't exist to help identify and isolate a class of error (*e. g.*, wild pointers), the error is more likely to survive undetected.

Unfixability. Risk of not being able to fix a bug.

Language-typical errors. Such as wild pointers in C. See for example, *Pitfalls of Object-Oriented Development* (Webster 1995) and *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs* (Daconta *et al*. 2000).

Use Error Catalogs

Testing Computer Software (Kaner *et al.* 1993) lays out a list of 480 common defects. You can use this list or develop your own. Here's how to use one:

1. Find a defect in the list.
2. Ask whether the software under test could have this defect.
3. If it's theoretically possible that the program could have the defect, ask how you could find the bug if it was there.
4. Ask how plausible it is that this bug could be in the program and how serious the failure would be if it was there.
5. If appropriate, design a test or series of tests for bugs of this type.

Unfortunately, too many people start and end with the TCS bug list. It's outdated. It was outdated the day it was published. And, it doesn't cover the issues in *your* system. Building a bug list is an ongoing process that

constantly pays for itself. Here's an example from Hung Nguyen (personal communication):

This problem came up in a client/server system. The system sends the client a list of names, to allow verification that a name the client enters is not new.

Clients 1 and 2 both want to enter a name, and Clients 1 and 2 both use the same new name. Both instances of the name are new relative to their local compare list and, therefore, they're accepted. We now have two instances of the same name.

As we see these, we develop a library of issues. The discovery method is exploratory and requires sophistication with the underlying technology. Capture winning themes for testing in charts or in scripts-on-their-way to being automated.

As you see new problems (within your time constraints), add them to your database to make them available to the testers on the next project.

Use Up-to-Date Sources to Add to Your Error Catalogs

There are plenty of sources to check for common failures in the common platforms. Here's a sample of the types of information available that we've found helpful:

www.bugnet.com

www.cnet.com

Nguyen (2001)

Telles and Hsieh (2001).