Black Box Software Testing

2004 Academic Edition

PART 3 -- DOMAIN TESTING

by

Cem Kaner, J.D., Ph.D.

Professor of Software Engineering

Florida Institute of Technology

and

James Bach

Principal, Satisfice Inc.

These notes are partially based on research that was supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Kaner & Bach grant permission to make digital or hard copies of this work for personal or classroom use, including use in commercial courses, provided that (a) Copies are not made or distributed outside of classroom use for profit or commercial advantage, (b) Copies bear this notice and full citation on the front page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion. Abstracting with credit is permitted. The proper citation for this work is "Black Box Software Testing (Course Notes, Academic Version, 2004) www.testingeducation.org", (c) Each page that you use from this work must bear the notice "Copyright (c) Cem Kaner and James Bach, kaner@kaner.com", or if you modify the page, "Modified slide, originally from Cem Kaner and James Bach", and (d) If a substantial portion of a course that you teach is derived from these notes, advertisements of that course should include the statement, "Partially based on materials provided by Cem Kaner and James Bach." To copy otherwise, to republish or post on servers, or to distribute to lists requires prior specific permission and a fee. Request permission to republish from Cem Kaner, kaner@kaner.com.

Domain Testing: Some Readings

Books with widely read introductions to domain testing

- Beizer, B. (1995) Black-Box Testing, John Wiley & Sons.
- Jorgensen, P.C. (2002) Software Testing: A Craftsman's Approach, 2d ed. CRC Press.
- Kaner, C., Falk, J., Nguyen, H.Q. (1993) Testing Computer Software, 2d ed. Van Nostrand Reinhold, reprinted John Wiley & Sons, 1999.
- Myers, G.J. (1979) The Art of Software Testing, John Wiley & Sons.
- Whittaker, J. (2002) How to Break Software, Addison-Wesley.

A few interesting papers. **These are required reading.**

- Clarke, L.A., Hassel, J. & Richardson, D.J. (1982) A close look at domain testing, IEEE Transactions on Software Engineering, v.2, p. 380-390.
- Kaner, C. (1998) Liability for Product Incompatibility. *Software QA,* Vol. 5, #4 (August/September), p. 33. http://www.kaner.com/pdfs/liability.pdf
- Kaner, C. (2004) Teaching domain testing: A status report, Proceedings of the Conference on Software Engineering Education & Training, *in press*. http://www.testingeducation.org/articles/domain_testing_cseet.pdf
- Ostrand,T. & Balcer, M. (1988, June) The category-partition method for specifying and generating functional tests. Communications of the ACM, Vol. 31 (6), pages 676-686.
- Hamlet, R. & Taylor, R. (1988, July) Partition Testing Does Not Inspire Confidence, Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, 206-215.
- Weyuker, E.J. & Jeng, B. (1991) Analyzing Partition Testing Strategies, IEEE Transactions on Software Engineering, v.17, p. 703-711.

 Copyright © 2004 Cem Kaner & James Bach

Notes on this Section

- Domain testing is the most commonly taught (and perhaps the most commonly used) software testing technique.
- We start our study of it in the traditional way that it is introduced to testing practitioners (for example, the way that it is introduced by Myers and by Kaner, Falk & Nguyen). That is, we develop the notion of equivalence classes and boundaries through careful analysis of a simple example, developing the idea all the way through the test documentation (boundary charts) traditionally recommended for this style of testing.
- To present a context for this technique (and the others we'll study), we also look at a typical sequence of tasks for starting the testing of a new application.
- In reality, domain testing isn't nearly this simple, and the simplistic descriptions may have done more harm than good by misleading testers into a belief that testing can be handled by a routine set of clearly defined procedures. ***We'll study some of the interesting complexities of domain testing in the next sections.***

Domain testing

- AKA partitioning, equivalence analysis, boundary analysis
- Fundamental question or goal:
  - This confronts the problem that there are too many test cases for anyone to run. This is a stratified sampling strategy that provides a rationale for selecting a few test cases from a huge population.
- General approach:
  - Divide the set of possible values of a field into subsets, pick values to represent each subset. The goal is to find a "best representative" for each subset, and to run tests with these representatives. Best representatives of ordered fields will typically be boundary values.
  - *Multiple variables*: combine tests of several "best representatives" and find a defensible way to sample from the set of combinations.
- Paradigmatic case(s)
  - Equivalence analysis of a simple numeric field.
  - Printer compatibility testing *(multidimensional variable, doesn't map to a simple numeric field, but stratified sampling is essential.)*

Let's work a simple example

Here is a program's specification:

- *This program is designed to add two numbers, which you will enter*
- *Each number should be one or two digits*
- *The program will print the sum. Press* `Enter` *after each number*
- *To start the program, type* `ADDER`

Before you start testing, do you have any questions about the spec?

» Refer to Testing Computer Software, Chapter 1, page 1

Working through the example

Here's my basic strategy for dealing with new code:

1. Start with obvious and simple tests. *Test the program with easy-to-pass values that will be taken as serious issues if the program fails.*
2. Test each function sympathetically. *Learn why this feature is valuable before you criticize it.*
3. Test broadly before deeply. *Check out all parts of the program quickly before focusing.*
4. Look for more powerful tests. *Once the program can survive the easy tests, put on your thinking cap and look systematically for challenges.*
5. Pick boundary conditions. *There will be too many good tests. You need a strategy for picking and choosing.*
6. Do some freestyle exploratory testing. *Run new tests every week, from the first week to the last week of the project.*

Refer to Testing Computer Software, Chapter 1

1. The simple, mainstream tests

For the first test, try a pair of easy values, such as 3 plus 7.

Here is the screen display that results from that test.

Are there any bug reports that you would file from this?

? **3**
? **7**
10

? _

Refer to Testing Computer Software, Chapter 1

2. Test each function sympathetically

- Why is this function here?
- What will the customer want to do with it?
- What is it about this function that, once it is working, will make the customer happy?
-

Knowing what the customer will want to do with the feature gives you a much stronger context for discovering and explaining what is wrong with the function, or with the function's interaction with the rest of the program.

3. Test broadly before deeply

– The objective of early testing is to flush out the big problems as quickly as possible.
– You will explore the program in more depth as it gets more stable.
– There is no point hammering a design into oblivion if it is going to change. Report as many problems as you think it will take to force a change, and then move on.

4. Look for more powerful tests

- Brainstorming Rules:
  - The goal is to get lots of ideas. You are brainstorming together to discover *categories* of possible tests.
  - There are more great ideas out there than you think.
  - Don't criticize others' contributions.
  - Jokes are OK, and are often valuable.
  - Work *later*, alone or in a much smaller group, to eliminate redundancy, cut bad ideas, and refine and optimize the specific tests.
  - Facilitator and recorder keep their opinions to themselves.

  We'll work more on brainstorming and, generally, on thinking in groups later.

Copyright © 2004 Cem Kaner & James Bach

4. Look for more powerful tests

What?	Why?
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________
____________________	____________________

Refer to Testing Computer Software, pages 4-5, for examples.

5. Reducing the testing burden

There are 199 x 199 = 39,601 test cases for valid values:

- 99 values: 1 to 99
- 1 value: 0
- 99 values: -99 to –1

____

199 values per variable

199 x 199 = 39,601 possible tests

So should we test them all?

We tested 3 + 7. Should we also test

- 4 + 7? 4 + 6?
- 2 + 7? 2 + 8?
- 3 + 8? 3 + 6?

Why?

5. Equivalence class & boundary analysis

- What about the values not in the spec?
  - 100 and above
  - -100 and below
  - anything non-numeric
- Should we run these tests?
  - Why?

5. Equivalence class & boundary analysis

- Some people want to automate these tests.
  - How would you automate them all?
  - How will you tell whether the program passed or failed?

We cannot afford to run every possible test. We need a method for choosing a few tests that will represent the rest. Equivalence analysis is the most widely used approach.

– refer to Testing Computer Software pages 4-5 and 125-132

5. Classical equivalence class and boundary analysis

- To avoid unnecessary testing, partition (divide) the range of inputs into groups of equivalent tests.
- Then treat an input value from the equivalence class as representative of the full group.
- We treat two tests as equivalent if they are so similar to each other that it seems pointless to test both.
- If you can map the input space to a number line, then boundaries mark the point or zone of transition from one equivalence class to another. These are good members of equivalence classes to use because the program is more likely to fail at a boundary.
  - Myers, Art of Software Testing, 45

These are fuzzy definitions of equivalence and boundary. We'll refine them soon.

5. Myers' boundary table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99 non-integer	99, 100 -99, -100 null entry 2.5	
Second number	same as first	same as first	same	

The traditional analysis would look at the potential numeric entries and partition them the way the specification would partition them.

5. Myers' boundary table (continued)

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99 non-integer non-number expressions	99, 100 -99, -100 null entry 0 2.5 / :	
Second number	same as first	same as first	same	

It might be useful to consider some other cases, such as special cases that are inside the range (e.g. 0) and errors on a different dimension from your basic "too big" and "too small".

Copyright © 2004 Cem Kaner & James Bach

5. Myers' boundary table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99 non-number expressions	99, 100 -99, -100 / : 0 null entry	
Second number	same as first	same as first	same	
Sum	-198 to 198			Are there other sources of data for this variable? Ways to feed it bad data?

We should also consider other variables, not just inputs. For example, think of output variables, interim results, variables as things that are stored, and variables as inputs to a subsequent process.

---See Whittaker, *How to Break Software*.

Boundary table as a test plan component

- Makes the reasoning obvious.
- Makes the relationships between test cases fairly obvious.
- Expected results are pretty obvious.
- Several tests on one page.
- Can delegate it and have tester check off what was done. Provides some limited opportunity for tracking.
- Not much room for status.

- Question, now that we have the table, must we do all the tests? What about doing them all each time (each cycle of testing)?

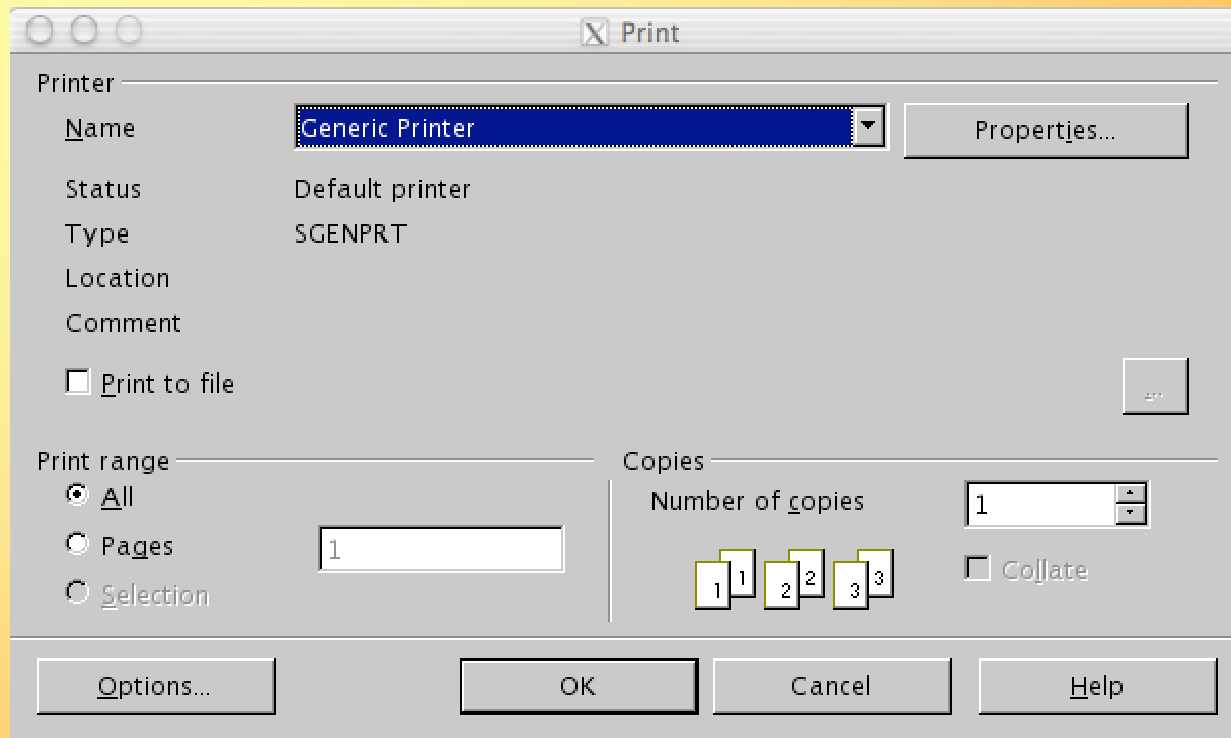
Building the table (in practice)

- Relatively few programs will come to you with all fields fully specified. Therefore, you should expect to learn what variables exist and their definitions over time.
- To build an equivalence class analysis over time, put the information into a spreadsheet. Start by listing variables. Add information about them as you obtain it.
- The table should eventually contain all variables. This means, all input variables, all output variables, and any intermediate variables that you can observe.
- In practice, most tables that I've seen are incomplete. The best ones that I've seen list all the variables and add detail for critical variables.

Review Question

- Gerald Weinberg's *Triangle Problem* has been in use since about 1969. Glen Myers published it in the first book on software testing, *The Art of Software Testing,* in 1979:
- The triangle program reads three numbers from a punch card (yes, that's right, ***a punch card***, so don't talk about what you'd do with some GUI) and interprets them as the sides of a triangle. The program then states whether the triangle is scalene, equilateral, or isosceles.
- How would you test this program? (List or describe your tests.)
- If this program was life-critical, what tests would you add? Why?

Simple Exercises



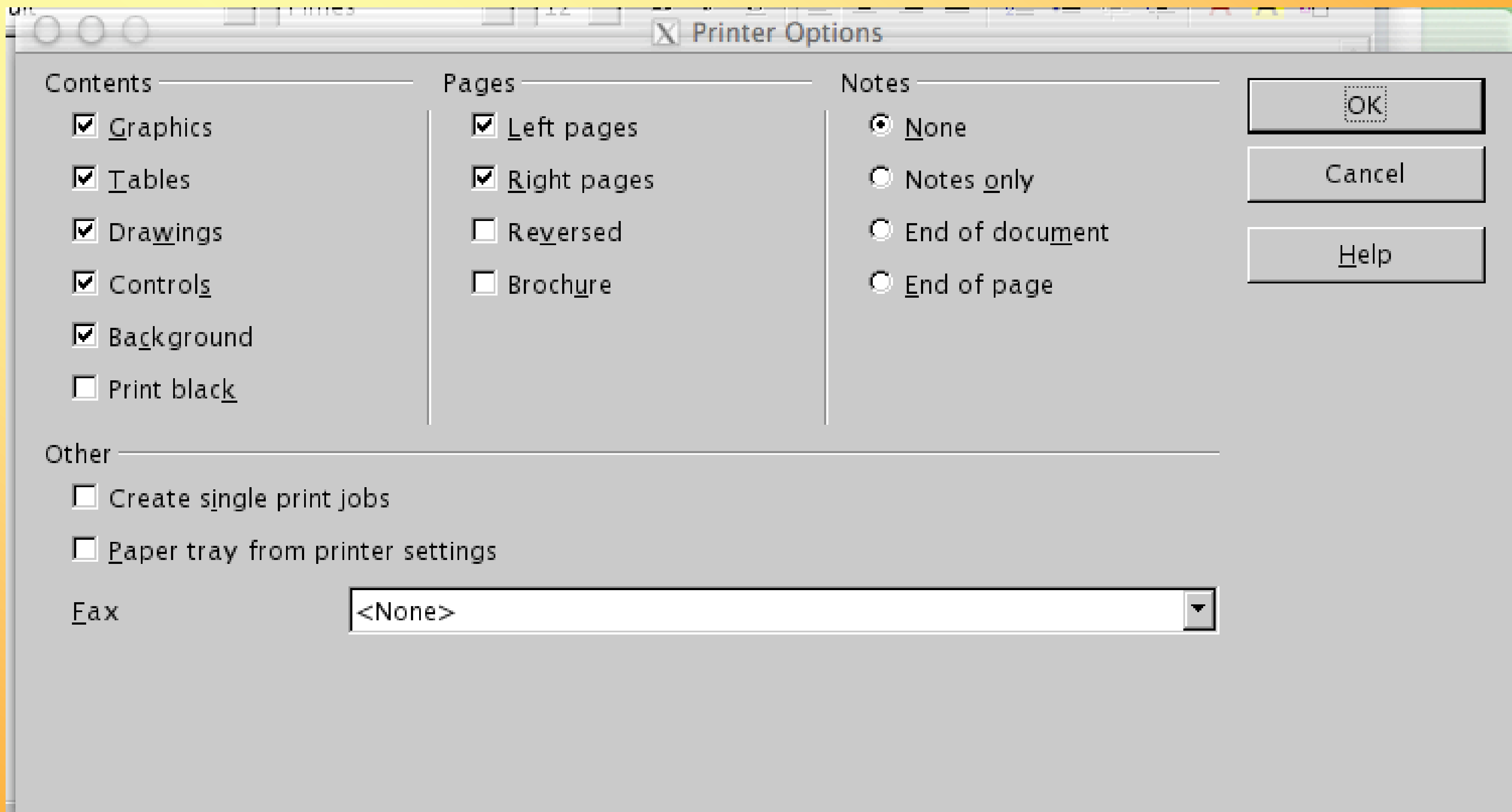
This is the print dialog in Open Office. Suppose that

1. The largest number of copies you could enter in Number of Copies field is 999, OR
2. Your printer will manage multiple copies, for up to 99 copies.

For each case, do a traditional domain analysis

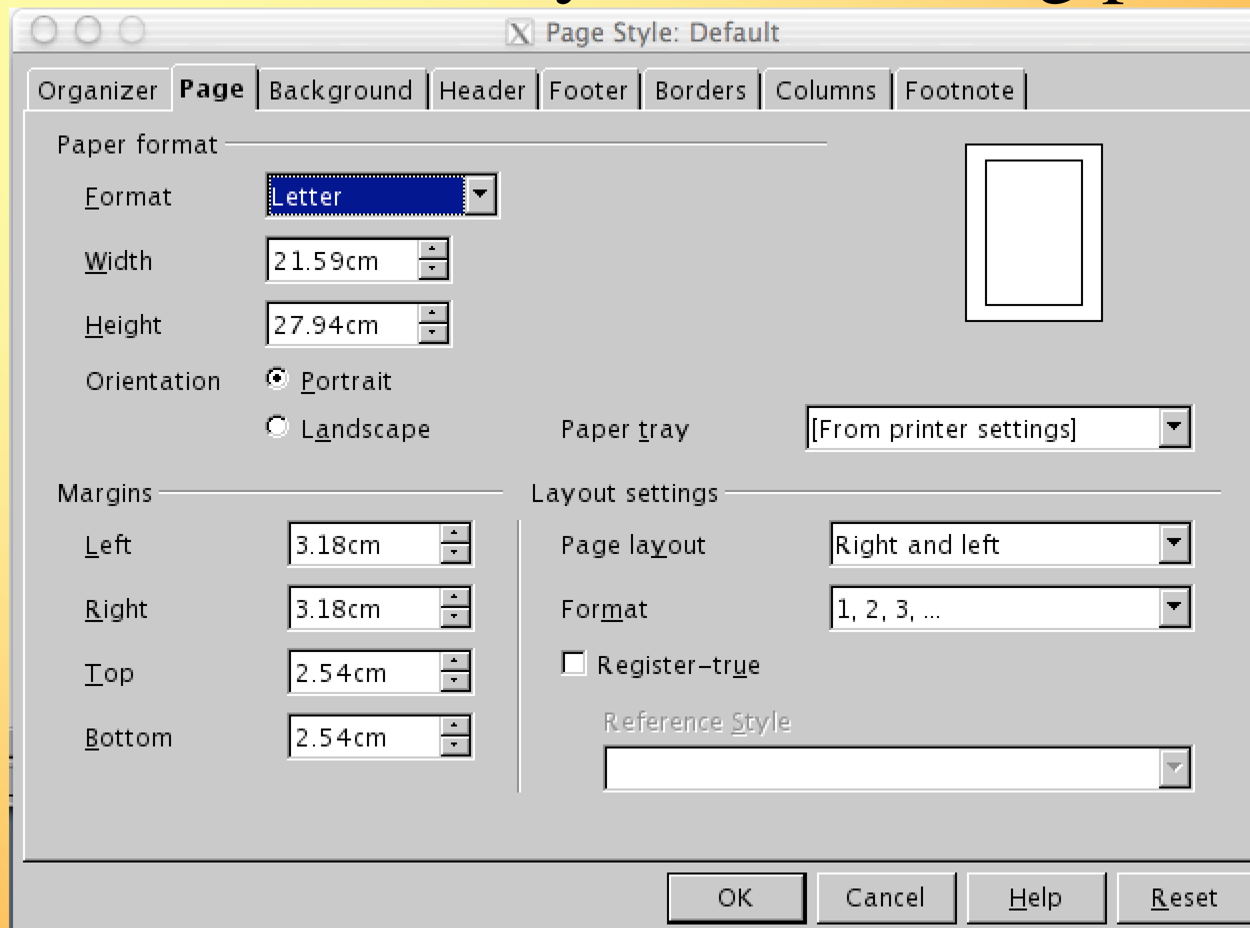
 Copyright © 2004 Cem Kaner & James Bach

Domain analysis on these variables?



- Would you do a domain analysis on these variables?
- What benefit would you gain from it?

Domain analysis on floating point



- Do a domain analysis on page width.
- What's the difference between this and analysis of an integer?

Myers' answer to the triangle problem

1. Test case for a *valid* scalene triangle
2. Test case for a valid equilateral triangle
3. Three test cases for valid isosceles triangles (a=b, b=c, a=c)
4. One, two or three sides has zero value (5 cases)
5. One side has a negative
6. Sum of two numbers equals the third (e.g. 1,2,3) is invalid b/c not a triangle (tried with 3 permutations a+b=c, a+c=b, b+c=a)
7. Sum of two numbers is less than the third (e.g. 1,2,4) (3 permutations)
8. Non-integer
9. Wrong number of values (too many, too few)

Examples of Myers' categories

1. {5,6,7}
2. {15,15,15}
3. {3,3,4; 5,6,6; 7,8,7}
4. {0,1,1; 2,0,2; 3,2,0; 0,0,9; 0,8,0; 11,0,0; 0,0,0}
5. {3,4,-6}
6. {1,2,3; 2,5,3; 7,4,3}
7. {1,2,4; 2,6,2; 8,4,2}
8. {Q,2,3}
9. {2,4; 4,5,5,6}

(examples courtesy of Doug Hoffman)

List 10 tests that you'd run that aren't in Myers' list.

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.