

Black Box Software Testing

2005 Academic Edition

DOMAIN TESTING

by

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

Professor of Software Engineering

Florida Institute of Technology

and

James Bach

Principal, Satisfice Inc.

Copyright (c) Cem Kaner & James Bach, 2000-2004

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

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.

Introductory Notes

- Domain testing is the most commonly taught (and perhaps the most commonly used) software testing technique.
- We start in the way it's traditionally introduced to testers (for example, 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.
- In practice, domain testing isn't nearly this simple. Simplistic descriptions may do 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 after our introduction.***

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 all parts of the program quickly before focusing.*
- 4 Look for more powerful tests. Try boundary conditions. *Once the program can survive the easy tests, you need a strategy for choosing powerful tests from the plethora of candidates.*
- 5 Expand your scope. *Put on your thinking cap; look for challenges.*
- 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. Classical equivalence & boundary analysis

There are 199 values for each variable:

1 to 99 99 values

0 1 value

-1 to -99 99 values

There are $199 \times 199 = 39,601$ combination tests

Should we test them all?

4. Classical equivalence class & boundary analysis

We tested $3 + 7$. Should we also test

$$4 + 7? \quad 4 + 6?$$

$$2 + 7? \quad 2 + 8?$$

$$3 + 8? \quad 3 + 6?$$

$$3 + 3? \quad 7 + 7?$$

Why? What would you learn from these?

What error would you expect one of these other tests to expose that $3+7$ would not already have exposed?

4. Classical equivalence class & boundary analysis

- What about the values not in the spec?
 - 100 and above
 - -100 and below
- Should we run these tests?
 - Why or why not?

4. Classical 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 powerful 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

4. Classical equivalence class & boundary analysis

- To avoid unnecessary testing, partition (divide) the range of inputs into groups of equivalent tests.
- We treat two tests as equivalent if they are so similar to each other that it seems pointless to test both.
- Select an input value from the equivalence class as representative of the full group.
- If you can map the input space to a number line, 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.

Myers' boundary table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Second number	-99 to 99	> 99 < -99	99, 100 -99, -100	

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

The classical boundary table

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Second number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Sum	-198 to 198	> 198 < -198	(-99,-99) (-99,99) (99,-99) (99,99)	Don't know how to create invalid-case tests

Combination tests of N variables create N-tuples with the boundary values of each of the component variables.

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.

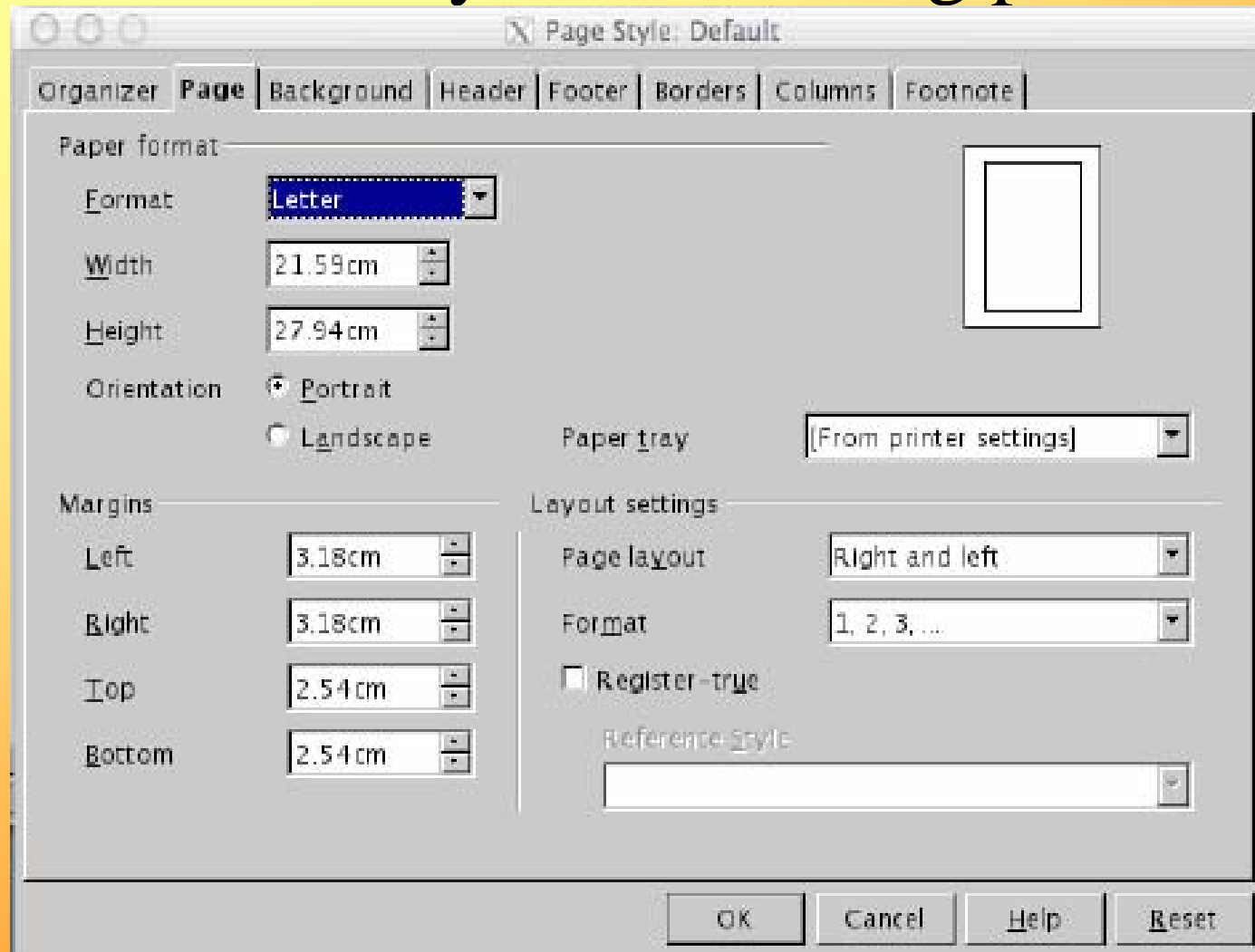
Scope of the analysis

- Several books stop here, or continue in the same direction, but look for ways to reduce the number of combination tests. See, for example:
 - Ilene Burnstein's *Practical Software Testing* (2004)
 - Paul Jorgensen's *Software Testing: A Craftsman's Approach* (2nd Ed., 2002)
 - Robert Binder's *Testing Object-Oriented Systems: Models, Patterns & Tools* (2000)
 - Boris Beizer's *Black Box Testing: Techniques for Functional Testing of Software & Systems* (1995)

What does this approach achieve?

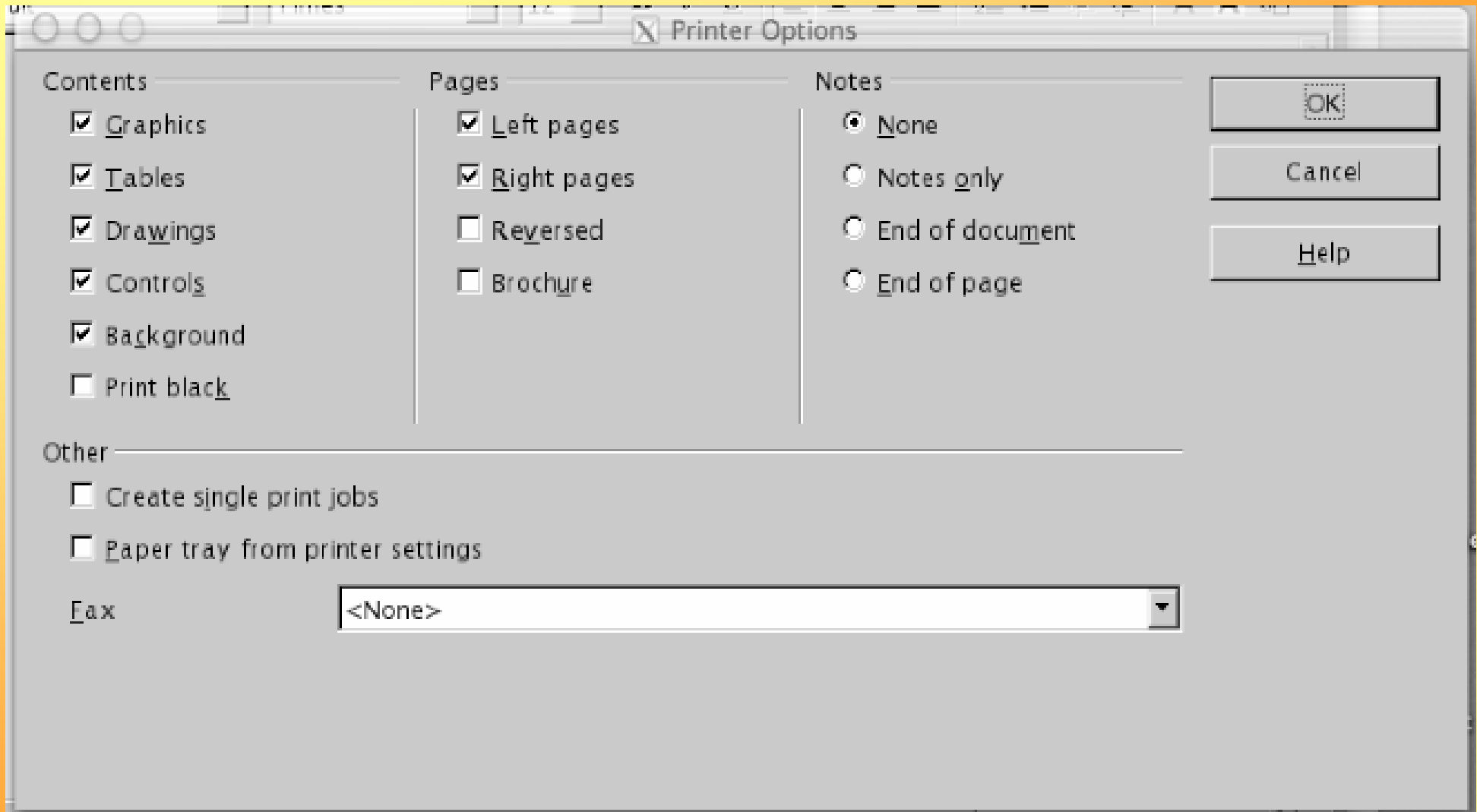
- This is a systematic sampling approach to test design. We can't afford to run all tests, so we divide the population of tests into subpopulations and test one or a few representatives of each subgroup. This keeps the number of tests manageable.
- Using *boundary values* for the tests offers a few benefits:
 - They will expose any errors that affect an entire equivalence class.
 - They will expose errors that mis-specify a boundary.
 - These can be coding errors (off-by-one errors such as saying “less than” instead of “less than or equal”) or typing mistakes (such as entering 57 instead of 75 as the constant that defines the boundary).
 - Mis-specification can also result from ambiguity or confusion about the decision rule that defines the boundary.
 - Non-boundary values are less likely to expose these errors.

Domain analysis on floating point



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

Domain analysis on these variables?



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

Examples of ordered sets

So many examples of domain analysis involve databases or simple data input fields that some testers don't generalize. Here's a sample of other variables that fit the traditional equivalence class / boundary analysis mold.

- ranges of numbers
- character codes
- how many times something is done
 - (e.g. shareware limit on number of uses of a product)
 - (e.g. how many times you can do it before you run out of memory)
- how many names in a mailing list, records in a database, variables in a spreadsheet, bookmarks, abbreviations
- size of the sum of variables, or of some other computed value (think binary and think digits)
- size of a number that you enter (number of digits) or size of a character string
- size of a concatenated string
- size of a path specification
- size of a file name
- size (in characters) of a document

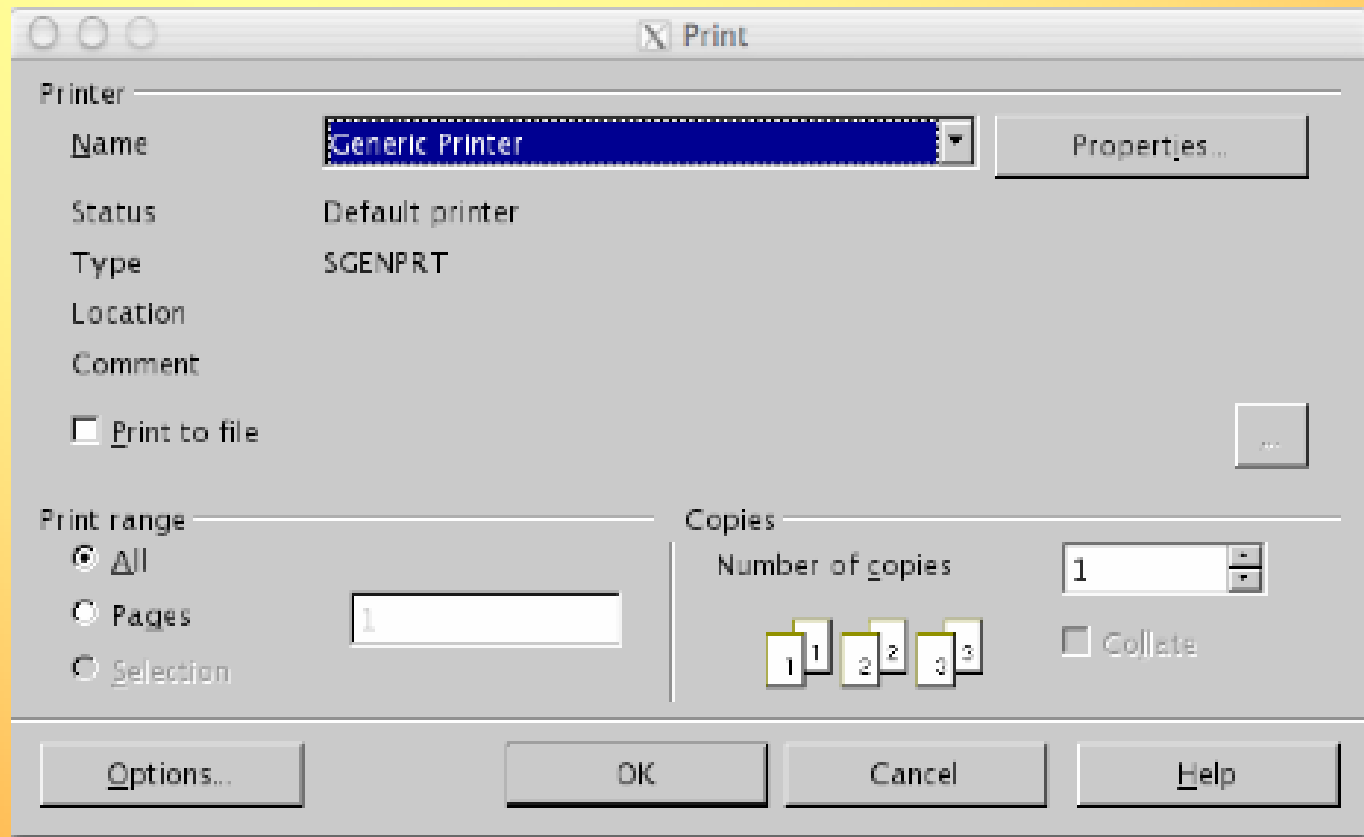
Examples of ordered sets

- size of a file (note special values such as exactly 64K, exactly 512 bytes, etc.)
- size of the document on the page (compared to page margins) (across different page margins, page sizes)
- size of a document on a page, in terms of the memory requirements for the page. This might just be in terms of resolution x page size, but it may be more complex if we have compression.
- equivalent output events (such as printing documents)
- amount of available memory (> 128 meg, > 640K, etc.)
- visual resolution, size of screen, number of colors
- operating system version
- variations within a group of “compatible” printers, sound cards, modems, etc.
- equivalent event times (when something happens)
- timing: how long between event A and event B (and in which order--races)
- length of time after a timeout (from JUST before to way after) -- what events are important?

Examples of ordered sets

- speed of data entry (time between keystrokes, menus, etc.)
- speed of input--handling of concurrent events
- number of devices connected / active
- system resources consumed / available (also, handles, stack space, etc.)
- date and time
- transitions between algorithms (optimizations) (different ways to compute a function)
- most recent event, first event
- input or output intensity (voltage)
- speed / extent of voltage transition (e.g. from very soft to very loud sound)

Domain analysis of results



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

Exercise

For each of the following,

- List the variable(s) of interest.
- List the valid and invalid classes.
- List the boundary value test cases.
- Lay out the results in a boundary table.

1. FoodVan delivers groceries to customers who order food over the Net. To decide whether to buy more vans, FV tracks the number of customers who call for a van. A clerk enters the number of calls into a database each day. Based on previous experience, the database is set to challenge (ask, “Are you sure?”) any number greater than 400 calls.

2. FoodVan schedules drivers one day in advance. To be eligible for an assignment, a driver must have special permission or she must have driven within 30 days of the shift she will be assigned to.

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?

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.

Extending the analysis

- Myers included other classes of examples:
 - Non-numeric values
 - Too few inputs or too many
 - Values that fit within the individual field constraints but that combine into an invalid result
- These are different in kind from tests that go after the wrong-boundary-specified error.
- Can we do boundary analysis on these?

Let's try it . . .

Potential error: Non-numeric values

	Character	ASCII Code
	/	47
lower bound	0	48
	1	49
	2	50
	3	51
	4	52
	5	53
	6	54
	7	55
	8	56
upper bound	9	57
	:	58
	A	65
	a	97

Refer to Testing
Computer Software,
pages 9-11

Potential error: Wrong number of inputs

- In the triangle example, the program wanted three inputs
- The valid class [of integers] is {3}
- The invalid classes [of integers] are
 - Any number less than 3 (boundary is 2)
 - Any number more than 3 (boundary is 4)

Potential error: Invalid combination

Consider these cases. Are these paired tests equivalent?

– If you tested

51+52

53+54

55+56

57+58

59+60

61+62

63+64

65+66

67+68

Would you test

52+53

54+55

56+57

58+59

60+61

62+63

64+65

66+67

68+69

Potential error: Invalid combination

Consider these cases. Are these paired tests equivalent?

– If you tested test

51+52

53+54

55+56

57+58

59+60

61+62

63+64

65+66

67+68

Would you

52+53

54+55

56+57

58+59

60+61

62+63

64+65

66+67

68+69

The hockey game example



- Earn 0 points for loss, 1 for tie, 2 for win. Sum of points stored in an unsigned integer
- Top teams go to the playoffs
- Up to 80 games—what if you win them all?

Potential error: Invalid combination

Consider these cases. Are these paired tests equivalent?

– If you tested test

51+52

53+54

55+56

57+58

59+60

61+62

63+64

65+66

67+68

Would you

52+53

54+55

56+57

58+59

60+61

62+63

64+65

66+67

68+69

The hockey game example



Once you've been burned by this integer overflow bug, 63+64 will never look the same as 64+65 again.

Another example of non-obvious boundaries

- Still in the 99+99 program
- Enter the first value
- Wait N seconds
- Enter the second value
- Suppose our client application will time out on input delays greater than 600 seconds. Does this affect how you would test?
- Suppose our client passes data that it receives to a server, the client has no timeout, and the server times out on delays greater than 300 seconds.
 - Would you discover this timeout from a path analysis of your application?
 - What boundary values should you test? In whose domains?

More examples of risks for the add-two-numbers example

- Memory corruption caused by input value too large.
- Failure on non-numeric input.
- Mishandles leading zeroes or leading spaces.
- Mishandles non-numbers inside number strings.
- Recovers poorly from its own error handling.
- Memory leaks.

5. Expand the scope

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.

Risk-based equivalence

- Given the following potential error:

These cases would not trigger the error, even if it was there.	These cases would trigger the error.

Extending the analysis

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	

I used to use Myers' table with an extended scope of errors and tests, but this hides the risks and the natural reasoning triggered by the risks

A new boundary / equivalence table

Variable	Risk (potential failure)	Classes that should not trigger the failure	Classes that might trigger the failure	Test cases (best representatives)	Notes
First input	Fail on out-of-range values	-99 to 99	MinInt to -100 100 to MaxInt	-100, 100	
	Doesn't correctly discriminate in-range from out-of-range			-100, -99, 100, 99	
	Misclassify digits	Non-digits	0 to 9	0 (ASCII 48) 9 (ASCII 57)	
	Misclassify non-digits	Digits 0 - 9	ASCII other than 48 - 57	/ (ASCII 47) ; (ASCII 58)	

Note that we've dropped the issue of "valid" and "invalid." This lets us generalize to partitioning strategies that don't have the **concept** of "valid" -- for example, **printer** equivalence classes. (For discussion of device compatibility testing, see Kaner et al., Chapter 8.)

Summary

- Domain analysis is a sampling strategy to cope with the problem of too many possible tests.
- Traditional domain analysis considers numeric input and output fields.
- Boundary analysis is optimized to expose a few types of errors such as miscoding of boundaries or ambiguity in definition of the valid/invalid sets.
 - However, there are other possible errors that boundary tests are insensitive to.
- Domain analysis often appears mechanical and routine. Given a numeric input field and its specified boundaries, we know what to do. But as we consider new risks, we have to add a new analysis and new tests.
- Rather than thinking we can pre-specify all the tests (after predicting all the risks), we should train testers in the application of equivalence classes to risk-based tests in general. As they discover new risks associated with a field (or with anything else) while testing, they can apply the analysis to come up with optimized new tests as needed.