Paradigms of Black Box Software Testing

Tutorial at Quality Week, 2002

Cem Kaner, J.D., Ph.D.
Florida Institute of Technology

kaner@kaner.com

www.kaner.com **www.badsoftware.com**

This material is based upon work supported by the National Science Foundation under Grant No. 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 (NSF).

About Cem Kaner

Professor of Computer Sciences, Florida Institute of Technology

Career focus--improving software customer satisfaction and safety. I've worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, development consultant, manager / director of almost all aspects of software development, and as an attorney focusing on the law of software quality.

Senior author of: *Testing Computer Software (*1988; 2nd edition with Hung Nguyen and Jack Falk,1993). This received the *Award of Excellence* in the Society for Technical Communication's Northern California Technical Publications Competition. *Bad Software: What To Do When Software Fails* (with David Pels, 1998). Ralph Nader called this book "a how-to book for consumer protection in the Information Age." and *Lessons Learned in Software Testing* (with James Bach & Bret Pettichord, 2001).

B.A. in Arts & Sciences (Math, Philosophy);

Ph.D. in Experimental Psychology (Human Perception & Performance: Psychophysics),

J.D. (law degree). Member of the American Law Institute.

Paradigms of Black Box Testing

ACKNOWLEDGEMENT

This section is based on work done jointly with James Bach, and uses slides that James and I prepared jointly for presentation at the USPDI and STAR conferences. We also thank Bob Stahl, Brian Marick, Hans Schaefer, and Hans Buwalda for several insights.

ABOUT JAMES BACH

- Principal of Satisfice, Inc., a software testing consulting company.
- Writes, speaks, and consults on software quality asssurance and testing. In his sixteen years in Silicon Valley-style software companies, including nine years at Apple Computer, Borland International, and Aurigin Systems, he's been a programmer, tester, QA manager, and roving problem-solver. For three years, James was Chief Scientist at ST Labs, an independent software testing company in Seattle that performs substantial testing and tester training for Microsoft.
- James is a frequent speaker and writer on the subject of software quality assurance and development processes. James edited the "Software Realities" column in *IEEE Computer* magazine, is a former section chair of the American Society for Quality and the Silicon Valley Software Quality Association, and was part of the ASQ committee that created the Certified Software Quality Engineer program.

Background Thoughts on Test Case Design

If the purpose of testing is to gain information about the product, then a test case's function is to elicit information quickly and efficiently.

In information theory, we define "information" in terms of reduction of uncertainty. If there is little uncertainty, there is little information to be gained.

A test case that promises no information is poorly designed. A good test case will provide information of value whether the program passes the test or fails it.

Dimensions of Test Case Quality

Think of a test case as a question you ask of the program. What are the attributes of the excellent question?

- Information
  - » *Reduction of uncertainty. How much do you expect to learn from this test?*
- Power
  - » *If two tests have the potential to expose the same type of error, one test is more powerful if it is more likely to expose the error.*
- Credibility
  - » *Failure of the test should motivate someone to fix it, not to dismiss it.*
- Feasibility
  - » *How hard is it to set up the test, run the test, and evaluate the result?*

Models

A Model is…

- A map of a territory
- A simplified perspective
- A relationship of ideas
- An incomplete representation of reality
- A diagram, list, outline, matrix…

No good test design has ever been done without models.

But models are often implicit, partially because of incomplete specs and conflicting stakeholder interests

A key goal is to become aware of how you model the product, and learn different ways of modeling.

The Puzzle

Black box testing groups vary widely in their approach to testing.

Tests that seem essential to one group seem uninteresting or irrelevant to another.

Big differences can appear even when both groups are composed of intelligent, experienced, dedicated people.

Why?

Paradigms

A paradigm provides a structure of thinking about an area.

- Typically, the description of a paradigm includes one or a few paradigmatic cases -- key example. Much of the reasoning within the paradigm is based on analogies to these key cases.
- The paradigm creates a mainstream of thinking--it provides insights and a direction for further research or work. But it implicitly also defines limits on what is relevant, interesting, or possible. Things outside of the paradigm are uninteresting. People who solve the types of puzzles that are characteristic of the paradigm are respected, whereas people who solve other types of puzzles instead are outsiders, and not well respected.
- *A testing paradigm would define the types of tests that are (to the person operating under this paradigm) relevant and interesting.*

Paradigms (Kuhn)

See Thomas Kuhn, *The Structure of Scientific Revolutions.* He describes paradigms as follows:

- **A paradigm is a model based on a shared experiment that includes law, theory, application and instrumentation together and from which springs particular coherent traditions of scientific research.**
- **A paradigm includes a concrete scientific achievement as a locus of professional commitment, prior to the various concepts, laws, theories and points abstracted from it.**
- *The pre-paradigm period . . . is regularly marked by frequent and deep debate over legitimate methods, problems, and standards of solution, though these serve rather to define schools than to produce agreement.*

Black Box Testing Paradigms

- **There are strikingly different paradigms (or sub-paradigms) *within* black box testing.**
- **This list reflects our (Kaner's/Bach's) observations in the field and is not exhaustive.**
- **We put one on the list if we've seen credible testers drive their thinking about black box testing in the way we describe. A paradigm for one person might merely be a technique for another.**
- ***We recommend that you try to master the "right" combination of two or three approaches. They are not mutually exclusive. The right combination will depend on your situation.***

A Quick Tour of the Paradigms

- Function testing
- Domain testing
- Specification-based testing
- Risk-based testing
- Stress testing
- Regression testing
- User testing
- Scenario testing
- Exploratory testing
- Stochastic or Random testing
- <<combination testing>>

Function Testing

Tag line

- "Black box unit testing."

Fundamental question or goal

- Test each function thoroughly, one at a time.

Paradigmatic case(s)

- Spreadsheet, test each item in isolation.
- Database, test each report in isolation

Strengths

- Thorough analysis of each item tested

Blind spots

- Misses interactions, misses exploration of the benefits offered by the program.

Domain Testing

Tag lines

- "Try ranges and options."
- "Subdivide the world into classes."

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. Typical values will be at boundaries. More generally, the goal is to find a "best representative" for each subset, and to run tests with these representatives.
- Advanced approach: combine tests of several "best representatives". Several approaches to choosing optimal small set of combinations.

Domain Testing

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.)*

- **In classical domain testing**
  - Two values (single points or n-tuples) are equivalent if the program would take the same path in response to each.
- **The classical domain strategies all assume**
  - that the predicate interpretations are simple, linear inequalities.
  - the input space is continuous and
  - coincidental correctness is disallowed.
- **It is possible to move away from these assumptions, but the cost can be high, and the emphasis on paths is troublesome because of the high number of possible paths through the program.**

Clarke, Hassell, & Richardson, p. 388

Domain Testing

Strengths

- Find highest probability errors with a relatively small set of tests.
- Intuitively clear approach, generalizes well

Blind spots

- Errors that are not at boundaries or in obvious special cases.
- Also, the actual domains are often unknowable.

Specification-Driven Testing

Tag line:

- "Verify every claim."

Fundamental question or goal

- Check the product's conformance with every statement in every spec, requirements document, etc.

Paradigmatic case(s)

- Traceability matrix, tracks test cases associated with each specification item.
- User documentation testing

Specification-Driven Testing

Strengths

- Critical defense against warranty claims, fraud charges, loss of credibility with customers.
- Effective for managing scope / expectations of regulatory-driven testing
- Reduces support costs / customer complaints by ensuring that no false or misleading representations are made to customers.

Blind spots

- Any issues not in the specs or treated badly in the specs /documentation.

Traceability Matrix

	Var 1	Var 2	Var 3	Var 4	Var 5
Test 1	X	X	X		
Test 2		X		X	
Test 3	X		X	X	
Test 4			X	X	
Test 5				X	X
Test 6	X				X

Traceability Matrix

The columns involve different test items. A test item might be a function, a variable, an assertion in a specification or requirements document, a device that must be tested, any item that must be shown to have been tested.

The rows are test cases.

The cells show which test case tests which items.

If a feature changes, you can quickly see which tests must be reanalyzed, probably rewritten.

In general, you can trace back from a given item of interest to the tests that cover it.

This doesn't specify the tests, it merely maps their coverage.

Risk-Based Testing

Tag line

- "Find big bugs first."

Fundamental question or goal

- Define and refine tests in terms of the kind of problem (or risk) that you are trying to manage

Paradigmatic case(s)

- Failure Mode and Effects Analysis (FMEA)
- Sample from predicted-bugs list.
- Stress tests, error handling tests, security tests, tests looking for predicted or feared errors.
- Equivalence class analysis, reformulated.

Risk-Based Testing

Strengths

- High power tests
- Intuitive tests

Blind spots

- Risks that were not identified or imagined

Stress Testing

Tag line

- "Overwhelm the product."
- "Drive it through failure."

Fundamental question or goal

- Learn about the capabilities and weaknesses of the product by driving it through failure and beyond. What does failure at extremes tell us about changes needed in the program's handling of normal cases?

Paradigmatic case(s)

- Buffer overflow bugs
- High volumes of data, device connections, long transaction chains
- Low memory conditions, device failures, viruses, other crises.

Strengths

- Expose weaknesses that will arise in the field.
- Expose security risks.

Blind spots

- Weaknesses that are not made more visible by stress.

Regression Testing

Tag line

- "Repeat testing after changes."

Fundamental question or goal

- Manage the risks that

  (a) a bug fix didn't fix the bug or

  (b) the fix (or other change) had a side effect.

Paradigmatic case(s)

- Bug regression (Show that a bug was not fixed)
- Old fix regression (Show that an old bug fix was broken)
- General functional regression (Show that a change caused a working area to break.)
- Automated GUI regression suites

Regression Testing

Strengths

- Reassuring, confidence building, regulator-friendly

Blind spots / weaknesses

- Anything not covered in the regression series.
- Repeating the same tests means not looking for the bugs that can be found by other tests.
- Pesticide paradox.
- Low yield from automated regression tests.
- Maintenance of this standard list can be costly and distracting from the search for defects.

Regression Automation

Regression tools dominate the automated testing market.

Why automate tests that the program has already passed?

- Percentage of bugs found with already-passed tests is about 5-20%
- Efficiency of regression testing shows up primarily in the next version or in a port to another platform.

If the goal is to find new bugs quickly and efficiently, we should use a method based on new test cases.

User Testing

Tag line

- Strive for realism
- Let's try this with real humans (for a change).

Fundamental question or goal

- Identify failures that will arise in the hands of a person, i.e. breakdowns in the overall human/machine/software system.

Paradigmatic case(s)

- Beta testing
- In-house experiments using a stratified sample of target market
- Usability testing

User Testing

Strengths

- Design issues are more credibly exposed.
- Can demonstrate that some aspects of product are incomprehensible or lead to high error rates in use.
- In-house tests can be monitored with flight recorders (capture/replay, video), debuggers, other tools.
- In-house tests can focus on areas / tasks that you think are (or should be) controversial.

Blind spots

- Coverage is not assured (serious misses from beta test, other user tests)
- Test cases can be poorly designed, trivial, unlikely to detect subtle errors.
- Beta testing is not free, beta testers are not skilled as testers, the technical results are mixed. Distinguish marketing betas from technical betas.

Scenario Testing

Tag lines

- "Do something useful and interesting"
- "Do one thing after another."

Fundamental question or goal

- Challenging cases that reflect real use.

Paradigmatic case(s)

- Appraise product against business rules, customer data, competitors' output
- Life history testing (Hans Buwalda's "soap opera testing.")
- Use cases are a simpler form, often derived from product capabilities and user model rather than from naturalistic observation of systems of this kind.

Scenario Testing

The ideal scenario has several characteristics:

- It is realistic (e.g. it comes from actual customer or competitor situations).
- There is no ambiguity about whether a test passed or failed.
- The test is complex, that is, it uses several features and functions.
- There is a stakeholder who has influence and will protest if the program doesn't pass this scenario.

Strengths

- Complex, realistic events. Can handle (help with) situations that are too complex to model.
- Exposes failures that occur (develop) over time

Blind spots

- Single function failures can make this test inefficient.
- Must think carefully to achieve good coverage.

Exploratory Testing

Tag line

- "Simultaneous learning, planning, and testing."

Fundamental question or goal

- Software comes to tester under-documented and/or late. Tester must simultaneously learn about the product and about the test cases / strategies that will reveal the product and its defects.

Paradigmatic case(s)

- Skilled exploratory testing of the full product
- Guerrilla testing (lightly but harshly test an area for a finite time)
- Rapid testing
- Emergency testing (including thrown-over-the-wall test-it-today testing.)
- Third party components.
- Troubleshooting / follow-up testing of defects.

Exploratory Testing

Simultaneously:

- Learn about the product
- Learn about the market
- Learn about the ways the product could fail
- Learn about the weaknesses of the product
- Learn about how to test the product
- Test the product
- Report the problems
- Advocate for repairs
- ***Develop new tests based on what you have learned so far.***

Exploratory Testing

Strengths

- Customer-focused, risk-focused
- Takes advantage of each tester's strengths
- Responsive to changing circumstances
- Well managed, it avoids duplicative analysis and testing
- High bug find rates

Blind spots

- The less we know, the more we risk missing.
- Limited by each tester's weaknesses (can mitigate this with careful management)
- This is skilled work, unsupervised juniors aren't very good at it.

Random / Statistical Testing

Tag line

- "High-volume testing with new cases all the time."

Fundamental question or goal

- Have the computer create, execute, and evaluate huge numbers of tests.
  - » The individual tests are not all that powerful, nor all that compelling.
  - » The power of the approach lies in the large number of tests.
  - » These broaden the sample, and they may test the program over a long period of time, giving us insight into longer term issues.

Random / Statistical Testing

Paradigmatic case(s)

- Oracle-driven, validate a function or subsystem (such as function equivalence testing)
- Stochastic (state transition) testing looking for specific failures (assertions, leaks, etc.)
- Statistical reliability estimation
- Partial or heuristic oracle, to find some types of errors without overall validation.

Strengths

- Regression doesn't depend on same old test every time.
- Partial oracles can find errors in young code quickly and cheaply.
- Less likely to miss internal optimizations that are invisible from outside.
- Can detect failures arising out of long, complex chains that would be hard to create as planned tests.

Random / Statistical Testing

Blind spots and Concerns

- Need to be able to distinguish pass from failure. Too many people think "Not crash = not fail."
- Also, these methods will often cover many types of risks, but will obscure the need for other tests that are not amenable to automation.
- Inappropriate prestige difference between automators and skilled manual testers

Combination Testing

I don't think combination testing is really a paradigm, but it involves a far-from-universal structured way of thinking about testing.

Tag line

- "Systematically and efficiently test variables in combination.

Fundamental question or goal

- To find some defects (or replicate some failures) you must control (consciously set) the values of two or more variables at the same time.
- Reduce a vast number of possible combinations to a sensible subset.

Combination Testing

Paradigmatic cases:

- Combining boundaries of several variables
- Combinatorial testing (e.g. all-pairs)
- Orthogonal arrays
- Cause-effect graphing

Contrast with scenario testing

- Coverage vs. credibility
- Systematic exploration of variable values vs. systematic exploration of market (stakeholder) concerns

Strengths

- If well done, broad combination coverage with relatively few cases

Concerns

- Tests are typically artificial, may not be credible

Paradigms and Techniques

The paradigms overlap. For example, you can do:

- Spec-based regression testing
- User scenario testing
- Risk-based automated GUI regression testing
- Exploratory domain testing

The difference lies in what's in the mind of the tester.

Paradigms and Techniques

A paradigm is a mental framework within which you design tests.

A technique is a recipe for creating tests. The design of any test case involves at least five dimensions. A technique specifies one or more of these dimensions:

- **Coverage:** What you're testing
- **Risks:** Problems you are trying to find
- **Evaluation:** How you know a test passed or failed
- **Activities:** How you test
- **Testers:** Who is testing

Any of the paradigms could describe a technique or family of techniques, but a "technique" user is likely to use several techniques, not rely primarily on one as the prototypic approach to testing.

Function Testing

Some Function Testing Tasks

Identify the program's features / commands

- From specifications or the draft user manual
- From walking through the user interface
- From trying commands at the command line
- From searching the program or resource files for command names

Identify variables used by the functions and test their boundaries.

Identify environmental variables that may constrain the function under test.

Use each function in a mainstream way (positive testing) and push it in as many ways as possible, as hard as possible.

Domain Testing

Domain Testing

Some of the Key Tasks

- Partitioning into equivalence classes
- Discovering best representatives of the sub-classes
- Combining tests of several fields
- Create boundary charts
- Find fields / variables / environmental conditions
- Identify constraints (non-independence) in the relationships among variables.

Domain Testing: The Classic Example

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

» From Glen Myers, *The Art of Software Testing*

Write a set of test cases that would adequately test this program.

Another Standard Example

Here is the 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

Brainstorm Test Ideas

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.

Brainstorm Test Ideas

What?	Why?

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

Domains: Brainstormed Test Ideas

There are too many tests.

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

- **definitely valid: 0 to 99**
- **might be valid: -99 to –1**

There are infinitely many invalid cases:

- **100 and above**
- **-100 and below**
- **anything non-numeric**

Some people want to automate these tests.

- **Can you run all the tests?**
- **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*

Domains: 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.
- Two tests are equivalent if we would expect that an error revealed by one would be revealed by the other.
- Boundaries mark the point or zone of transition from one equivalence class to another. The program is more likely to fail at a boundary, so these are good members of equivalence classes to use.

Myers, Art of Software Testing, 45

Domains: Back to the Test Ideas

For each test idea on the brainstormed list:

- What class does this belong to?
  - » How can we decide what class? ***Well, what kind of defect is this test well suited to detect?***
- What other members of the class are there?
  - » ***What other tests are well suited to detect this type of defect?***
- Which tests in this class are the most powerful -- most likely to detect this type of defect?
  - » ***These are your best representatives. Often, but not always, they are boundary cases.***
- ***Note the broad range of test ideas, associated with the broad range of risks.***

Expanding the Notion of Equivalence

Consider these cases. Are these paired tests equivalent?

(55+56, 56+57)	**(57+58, 58+59)**
(59+60, 60+61)	**(61+62, 62+63)**
(63+64, 64+65)	**(65+66, 66+67)**
(67+68, 68+69)	**(69+70, 70+71)**

Example: The Hockey Game

Another Example: Boundaries May Not Be Obvious

	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

Equivalence Classes and Partitions are Subjective.

My working definition of equivalence:

> *Two test cases are equivalent if you expect the same result from each.*

This is fundamentally subjective. It depends on what you expect. And what you expect depends on what errors you can anticipate:

> *Two test cases can only be equivalent by reference to a specifiable risk.*

Two different testers will have different theories about how programs can fail, and therefore they will come up with different classes.

Equivalence Classes and Boundaries

Two tests belong to the same *equivalence class* if you expect the same result (pass / fail) of each. Testing multiple members of the same equivalence class is, by definition, redundant testing.

***Boundaries* mark the point or zone of transition from one equivalence class to another. The program is more likely to fail at a boundary, so these are the best members of (simple, numeric) equivalence classes to use.**

Note how the boundary case has two ways to fail. It can fail because the program's treatment of the equivalence class is broken OR because the programmer's treatment of inequalities is broken.

More generally, you look to subdivide a space of possible tests into relatively few classes and to run a few cases of each. You'd like to pick the most powerful tests from each class.

Boundary Analysis Table

Variable	Equivalence Class	Alternate Equivalence Class	Boundaries and Special Cases	Notes
First number	-99 to 99 digits	> 99 < -99 non-digits expressions	99, 100 -99, -100 /, 0, 9, : leading spaces or 0s null entry	
2nd number	same as first	same as first	same	
Sum	-198 to 198 -127 to 127	??? -198 to –128 128 to 198	??? 127, 128, -127, -128	Are there other sources of data for this variable? Ways to feed it bad data?

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.)

Domain Testing

In classical domain testing

- Two values (single points or n-tuples) are equivalent if the program would take the same path in response to each.

The classical domain strategies all assume

- that the predicate interpretations are simple, linear inequalities.
- the input space is continuous and
- coincidental correctness is disallowed.

"It is possible to move away from these assumptions, but the cost can be high. The emphasis on paths is troublesome because of the high number of possible paths through the program."

Clarke, Hassell, & Richardson, p. 388

Exercise

For each of the following, list

- The variable(s) of interest
- The valid and invalid classes
- The boundary value test cases.

1. FoodVan delivers groceries to customers who order food over the Net. To decide whether to buy more 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.

Domains: Notes on Myers' Exercise

Several classes of issues were missed by most students. For example:

- Few students checked whether they were producing valid triangles. (1,2,3) and (1,2,4) cannot be the lengths of any triangle.
  - » *Knowledge of the subject matter of the program under test will enable you to create test cases that are not directly suggested by the specification. If you lack that knowledge, you will miss key tests.* **(This knowledge is sometimes called "domain knowledge", not to be confused with "domain testing.")**
- Few students checked non-numeric values, bad delimiters, or non-integers.
- The only boundaries tested were at MaxInt or 0.

Domains: Myers' Answer

- **Test case for a *valid* scalene triangle**
- **Test case for a valid equilateral triangle**
- **Three test cases for valid isosceles triangles (a=b, b=c, a=c)**
- **One, two or three sides has zero value (5 cases)**
- **One side has a negative**
- **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)**
- **Sum of two numbers is less than the third (e.g. 1,2,4) (3 permutations)**
- **Non-integer**
- **Wrong number of values (too many, too few)**

Domain Testing

Some Relevant Skills

- Identify ambiguities in specifications or descriptions of fields
- Find biggest / smallest values of a field
- Discover common and distinguishing characteristics of multi-dimensional fields, that would justify classifying some values as "equivalent" to each other and different from other groups of values.
- Standard variable combination methods, such as all-pairs or the approaches in Jorgensen and Beizer's books

Domain Testing

Practice exercises at home

- Find the biggest / smallest accepted value in a field
- Find the biggest / smallest value that fits in a field
- Partition fields
- Read specifications to determine the actual boundaries
- Create boundary charts for several variables
- Create standard domain testing charts for different types of variables
- For finding variables, see notes on function testing

Domains: Notes on Myers' Exercise

Several classes of issues were missed by most students. For example:

- Few students checked whether they were producing valid triangles. (1,2,3) and (1,2,4) cannot be the lengths of any triangle.
  - » *Knowledge of the subject matter of the program under test will enable you to create test cases that are not directly suggested by the specification. If you lack that knowledge, you will miss key tests.* **(This knowledge is sometimes called "domain knowledge", not to be confused with "domain testing.")**
- Few students checked non-numeric values, bad delimiters, or non-integers.
- The only boundaries tested were at MaxInt or 0.

Domains: Myers' Answer

- **Test case for a *valid* scalene triangle**
- **Test case for a valid equilateral triangle**
- **Three test cases for valid isosceles triangles (a=b, b=c, a=c)**
- **One, two or three sides has zero value (5 cases)**
- **One side has a negative**
- **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)**
- **Sum of two numbers is less than the third (e.g. 1,2,4) (3 permutations)**
- **Non-integer**
- **Wrong number of values (too many, too few)**

Additional Basic Exercises

Analyze some common dialog boxes, such as these in MS Word:

- Print dialog
- Page setup dialog
- Font format dialog

For each dialog

- Identify each field, and for each field
  - » Name the type of the field (integer, rational, string, etc.)
  - » List the range of entries that are "valid" for the field
  - » Partition the field and identify boundary conditions
  - » List the entries that are almost too extreme and too extreme for the field
  - » List a small number of test cases for the field and explain why the values you have chosen are best representatives of the sets of values that they were selected from.
  - » Identify any constraints imposed on this field by other fields

Domain Testing: Interesting Papers

- Thomas Ostrand & Mark Balcer, *The Category-partition Method For Specifying And Generating Functional Tests,* Communications of the ACM, Vol. 31, No. 6, 1988.
- Debra Richardson, et al., *A Close Look at Domain Testing*, IEEE Transactions On Software Engineering, Vol. SE-8, NO. 4, July 1982
- Michael Deck and James Whittaker**,** *Lessons learned from fifteen years of cleanroom testing. STAR '97 Proceedings* (in this paper, the authors adopt boundary testing as an adjunct to random sampling.)
- Richard Hamlet & Ross Taylor, *Partition Testing Does Not Inspire Confidence*, Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, 206-215, July 1988

A Three-Variable Boundary Example

I, J, and K are integers. The program calculates K = I*J. For this question, consider only cases in which you enter integer values into I and J. Do an equivalence class analysis from the point of view of the effects of I and J (jointly) on the variable K. Identify the boundary tests that you would run (the values you would enter into I and J) if

- I, J, K are unsigned integers
- I,J, K are signed integers

Interaction example

K can run from MinInt (smallest integer) to MaxInt.

For any value of K, there is a set of values of I and J that will yield K

- The set is the null set for impossible values of K
- The set might include only two pairs, such as K = MaxInt, when MaxInt is prime (7 could be a MaxInt)
  - » (I,J) Є {(1, MaxInt), (MaxInt, 1)}
- The set might include a huge number of pairs, such as when K is 0:
  - » (I,J) Є {(0,0), (1, 0), (2, 0), … (MaxInt,0), (0,1),…,(0, MaxInt,1)}

- **A set of pairs of values of I and J can be thought of as an equivalence set (they all yield the same value of K) and so we ask which values of K are interesting (partition number 1) and then which values of I and J would produce those K-values, and do a partitioning on the sets of (I,J) pairs to find best representatives of those.**
- **As practitioners, we do this type of analysis often, but many of us probably don't think very formally about it.**

Fuzzy Boundaries

In theory, the key to partitioning is dividing the space into mutually exclusive ranges (subsets). Each subset is an equivalence class. This is nice in theory, but let's look at printers.

Problem:

- There are about 2000 Windows-compatible printers, plus multiple drivers for each. We can't test them all.

But maybe we can form equivalence classes.

An example from two programs (desktop publishing and an address book) developed in 1991-92.

Fuzzy Boundaries

Primary groups of printers at that time:

- HP - Original
- HP - LJ II
- PostScript Level I
- PostScript Level II
- Epson 9-pin, etc.

LaserJet II compatible printers, huge class (maybe 300 printers, depending on how we define it)

1. Should the class include LJII, LJII+, and LIIP, LJIID-compatible subclasses?
2. What is the best representative of the class?

Fuzzy Boundaries

Example: graphic complexity error handling

- HP II original was the weak case.

Example: special forms

- HP II original was strong in paper-handling. We worked with printers that were weaker in paper-handling.

Examples of additional queries for almost-equivalent printers

- Same margins, offsets on new printer as on HP II original?
- Same printable area?
- Same handling of hairlines? (Postscript printers differ.)

What exercises can we do to support development of this type of analysis?

Partitioning

Device compatibility testing illustrates a multidimensional space with imperfect divisions between classes and with several different failure risks. From an equivalence class of "LaserJet II compatibles" you get several different, uniquely powerful, class representatives.

The key to success is to remember that PARTITIONING IS MERELY A SAMPLING STRATEGY. The goal is to work from a rational basis in order to select a few valuable representatives from a much larger population of potential tests.

A strong sampling strategy rests on our knowledge of the world, not just of the specification.

If you can think of different ways that the program can fail in its interaction with a device (such as a printer), then FOR EACH TYPE OF ERROR, you look for the specific device (model, version of printer) that is most likely to confound the program.

We might miss testing a key device because we didn't think through all of the failure modes we were looking for.

Partitioning

The problem (as laid out by a client):

> "Equivalence class methods would be very valuable in situations where someone wants every OEM system, every sound and video card, every operating system, multiple technologies (e.g., DirectX 3 and 5), etc.
>
> "How can the engineer be confident in developing an equivalence matrix that provides good coverage?"

The disappointing but very real answer:

> "Despite good analysis and execution, we might simply miss a device or driver or a combination of devices and drivers that has a bug associated with it. If we convince a stakeholder that equivalence class analysis provides "complete" coverage, she will be justifiably upset if our 'complete coverage' misses a bug."

Equivalence Classes: A Broad Concept

The notion of equivalence class is much broader than numeric ranges. Here are some examples:

- Membership in a common group
  - » such as employees vs. non-employees. (Note that not all classes have shared boundaries.)
- Equivalent hardware
  - » such as compatible modems
- Equivalent event times
  - » such as before-timeout and after
- Equivalent output events
  - » perhaps any report will do to answer a simple the question: Will the program print reports?
- Equivalent operating environments
  - » such as French & English versions of Windows 3.1

Variables Suited to Equivalence Class Analysis

There are many types of variables, such as:

- input variables
- output variables
- internal variables
- hardware and system software configurations, and
- equipment states.

Any of these can be subject to equivalence class analysis.

Here are some examples:

Variables Well Suited to Equivalence Class Analysis

There are many types of variables, including input variables, output variables, internal variables, hardware and system software configurations, and equipment states. Any of these can be subject to equivalence class analysis. Here are some examples:

- **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**

Variables Well Suited to Equivalence Class Analysis

- 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?

Variables Well Suited to Equivalence Class Analysis

- **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)**

Closing Notes on Boundary Analysis

- In the traditional approach, we analyzed the code from the point of view of the programmer and the specification.
- The specification provides us with only one of several classes of risk. We have to test against a broader set.
- Experts in different subject matters will see different opportunities for failure, and different classes of cases that might reveal these failures.
- Thinking outside of the box posed by the explicit design is at the essence of black box testing.

Reusable Test Matrices (and Domain Testing)

Using Test Matrices for Routine Issues

After testing a simple numeric input field a few times, you've learned the drill. The boundary chart is reasonably easy to fill out for this, but it wastes your time.

Use a test matrix to show/track a series of test cases that are essentially the same.

- For example, for most input fields, you'll do a series of the same tests, checking how the field handles boundaries, unexpected characters, function keys, etc.
- As another example, for most files, you'll run essentially the same tests on file handling.

The matrix is a concise way of showing the repeating tests.

- Put the objects that you're testing on the rows.
- Show the tests on the columns.
- Check off the tests that you actually completed in the cells.

Typical Uses of Test Matrices

- You can often re-use a matrix like this across products and projects.
- You can create matrices like this for a wide range of problems. Whenever you can specify multiple tests to be done on one class of object, and you expect to test several such objects, you can put the multiple tests on the matrix.
- Mark a cell green if you ran the test and the program passed it.
- Mark the cell red if the program failed and write the bug number of the bug report for this bug.
- Write (in the cell) the automation number or identifier or file name if the test case has been automated.

Matrices

Problems?

- **What if your thinking gets out of date? (What if this program poses new issues, not covered by the standard tests?)**
- **Do you need to execute every test every time? (or ever?)**
- **What if the automation ID number changes? -- We still have a maintenance problem but it is not as obscure.**

Reusable Test Matrix

Numeric (Integer) Input Field													
		Nothing	LB of value	UB of value	LB of value - 1	UB of value + 1	0	Negative	LB number of digits or chars	UB number of digits or chars	Empty field (clear the default value)	Outside of UB number of digits or chars	Non-digits

This includes only the first few columns of a matrix that I've used commercially, but it gets across the idea.

Examples of integer-input tests

Nothing
Valid value
At LB of value
At UB of value
At LB of value - 1
At UB of value + 1
Outside of LB of value
Outside of UB of value
0
Negative
At LB number of digits or chars
At UB number of digits or chars
Empty field (clear the default value)

Outside of UB number of digits or chars
Non-digits
Wrong data type (e.g. decimal into integer)
Expressions
Space
Non-printing char (e.g., Ctrl+char)
DOS filename reserved chars (e.g., "\ * . :")
Upper ASCII (128-254)
Upper case chars
Lower case chars
Modifiers (e.g., Ctrl, Alt, Shift-Ctrl, etc.)
Function key (F2, F3, F4, etc.)

Error Handling when Writing a File

- full local disk
- almost full local disk
- write protected local disk
- damaged (I/O error) local disk
- unformatted local disk
- remove local disk from drive after opening file
- timeout waiting for local disk to come back online
- keyboard and mouse I/O during save to local disk
- other interrupt during save to local drive
- power out during save to local drive
- full network disk
- almost full network disk
- write protected network disk
- damaged (I/O error) network disk
- remove network disk after opening file
- timeout waiting for network disk
- keyboard / mouse I/O during save to network disk
- other interrupt during save to network drive
- local power out during save to network
- network power during save to network

Specification-Based Testing

Specification

Tasks

- review specifications for
  - » Ambiguity
  - » Adequacy (it covers the issues)
  - » Correctness (it describes the program)
  - » Content (not a source of design errors)
  - » Testability support
- Create traceability matrices
- Document management (spec versions, file comparison utilities for comparing two spec versions, etc.)
- Participate in review meetings

Specification

Skills

- Understand the level of generality called for when testing a spec item. For example, imagine a field X:
  - » We could test a single use of X
  - » Or we could partition possible values of X and test boundary values
  - » Or we could test X in various scenarios
  - » Which is the right one?
- Ambiguity analysis
  - » Richard Bender teaches this well. If you can't take his course, you can find notes based on his work in Rodney Wilson's **Software RX: Secrets of Engineering Quality Software**

Specification

Skills

- Ambiguity analysis
  - » Another book provides an excellent introduction to the ways in which statements can be ambiguous and provides lots of sample exercises: Cecile Cyrul Spector, *Saying One Thing, Meaning Another : Activities for Clarifying Ambiguous Language*

Breaking Statements into Elements

Make / read a statement about the program

Work through the statement one word at a time, asking what each word means or implies.

- *Thinkertoys* describes this as "slice and dice."
- *Gause & Weinberg* develop related approaches as
  - » "Mary had a little lamb" (read the statement several times, emphasizing a different word each time and asking what the statement means, read that way)
  - » "Mary conned the trader" (for each word in the statement, substitute a wide range of synonyms and review the resulting meaning of the statement.)
  - » These approaches can help you ferret out ambiguity in the definition of the product. By seeing how different people could interpret a key statement (e.g. spec statement that defines part of the product), you can see new test cases to check which meaning is operative in the program.

Breaking Statements into Elements: An Example

Quality is value to some person

- Quality
  - »
  - »
  - »
- Value
  - »
  - »
  - »
- Some
  - »
  - »
  - »
- Person
  - »

- *Who is this person?*
- *How are you the agent for this person?*
- *How are you going to find out what this person wants?*
- *How will you report results back to this person?*
- *How will you take action if this person is mentally absent?*

Reviewing a Specification for Completeness

Reading a spec linearly is not a particularly effective way to read the document. It's too easy to overlook key missing issues.

We don't have time to walk through this method in this class, but the general approach that I use is based on James Bach's "Satisfice Heuristic Test Strategy Model" at http://www.satisfice.com/tools/satisfice-tsm-4p.pdf.

- You can assume (not always correctly, but usually) that every sentence in the spec is meant to convey information.
- The information will probably be about
  - » the project and how it is structured, funded or timed, or
  - » about the product (what it is and how it works) or
  - » about the quality criteria that you should evaluate the product against.

Reviewing a Specification for Completeness

Spec Review using the Satisfice Model, continued

- The Satisfice Model lists several examples of project factors, product elements and quality criteria.
- For a given sentence in the spec, ask whether it is telling you project, product, or quality-related information. Then ask whether you are getting the full story. As you do the review, you'll discover that project factors are missing (such as deadline dates, location of key files, etc.) or that you don't understand / recognize certain product elements, or that you don't know how to tell whether the program will satisfy a given quality criterion.
- Write down these issues. These are primary material for asking the programmer or product manager about the spec.

Getting Information When There Is No Specification

(Suggestions from some brainstorming sessions.)

- Whatever specs exist
- Software change memos that come with each new internal version of the program
- User manual draft (and previous version's manual)
- Product literature
- Published style guide and UI standards
- Published standards (such as C-language)
- 3rd party product compatibility test suites
- Published regulations
- Internal memos (e.g. project mgr. to engineers, describing the feature definitions)
- Marketing presentations, selling the concept of the product to management
- Bug reports (responses to them)
- Reverse engineer the program.
- Interview people, such as
  - development lead
  - tech writer
  - customer service
  - subject matter experts
  - project manager
- Look at header files, source code, database table definitions
- Specs and bug lists for all 3rd party tools that you use
- Prototypes, and lab notes on the prototypes

Getting Information When There Is No Specification

- Interview development staff from the last version.
- Look at customer call records from the previous version. What bugs were found in the field?
- Usability test results
- Beta test results
- Ziff-Davis SOS CD and other tech support CD's, for bugs in your product and common bugs in your niche or on your platform
- BugNet magazine / web site for common bugs
- News Groups, CompuServe Fora, etc., looking for reports of bugs in your product and other products, and for discussions of how some features are supposed (by some) to work.
- Localization guide (probably one that is published, for localizing products on your platform.)
- Get lists of compatible equipment and environments from Marketing (in theory, at least.)
- Look at compatible products, to find their failures (then look for these in your product), how they designed features that you don't understand, and how they explain their design. See listserv's, NEWS, BugNet, etc.
- Exact comparisons with products you emulate
- Content reference materials (e.g. an atlas to check your on-line geography program)

Risk-Based Testing

Equivalence and Risk

Our working definition of equivalence:

> *Two test cases are equivalent if you expect the same result from each.*

This is fundamentally subjective. It depends on what you expect. And what you expect depends on what errors you can anticipate:

> *Two test cases can only be equivalent by reference to a specifiable risk.*

Two different testers will have different theories about how programs can fail, and therefore they will come up with different classes.

A boundary case in this system is a "best representative."

> *A best representative of an equivalence class is a test that is at least as likely to expose a fault as every other member of the class.*

Evaluating Risk

- **Several approaches that call themselves "risk-based testing" ask which tests we should run and which we should skip if we run out of time.**
- **These seem more like risk-based test management. I'd rather focus on test design.**
  - If a key purpose of testing is to find defects, a key strategy for testing should be defect-based. Every test should be questioned:
    - » How will this test find a defect?
    - » What kind of defect do you have in mind?
    - » What power does this test have against that kind of defect? Is there a more powerful test? A more powerful suite of tests?

Risk-Based Testing

Many of us who think about testing in terms of risk, analogize testing of software to the testing of theories:

- Karl Popper, in his famous essay *Conjectures and Refutations*, lays out the proposition that a scientific theory gains credibility by being subjected to (and passing) harsh tests that are intended to refute the theory.
- We can gain confidence in a program by testing it harshly (if it passes the tests).
- Subjecting a program to easy tests doesn't tell us much about what will happen to the program in the field.

In risk-based testing, we create harsh tests for vulnerable areas of the program.

Risk-Based Testing

Tasks

- Identify risk factors (hazards: ways in which the program could go wrong)
- For each risk factor, create tests that have power against it.
- Assess coverage of the testing effort program, given a set of risk-based tests. Find holes in the testing effort.
- Build lists of bug histories, configuration problems, tech support requests and obvious customer confusions.
- Evaluate a series of tests to determine what risk they are testing for and whether more powerful variants can be created.

Risk-Based Testing: Definitions

- Hazard:
  - » A dangerous condition (something that could trigger an accident)
- Risk:
  - » Possibility of suffering loss or harm (probability of an accident caused by a given hazard).
- Accident:
  - » A hazard is encountered, resulting in loss or harm.

Risks: Where to look for errors

New things**: newer features may fail.**

New technology: **new concepts lead to new mistakes.**

Learning Curve: **mistakes due to 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.**

Tired programmers: **long overtime over several weeks or months yields inefficiencies and errors**

Adapted from James Bach's lecture notes

Risks: Where to look for errors

Other staff issues: **alcoholic, mother died, two programmers who won't talk to each other (neither will their code)…**

Just slipping it in: **pet feature not on plan may interact badly with other code.**

N.I.H.: **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.**

Adapted from James Bach's lecture notes

Risks: Where to look for errors

Conflicting requirements: **ambiguity often hides conflict, result is 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 contract but fail product. (check out** http//www.agilealliance.org/)

Complexity**: complex code may be buggy.**

Bugginess: **features with many known bugs may also have many unknown bugs.**

Adapted from James Bach's lecture notes

Risks: Where to look for errors

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: **(e.g. regression, function tests), can yield a backlog of errors surviving across versions.**

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

Adapted from James Bach's lecture notes

Risks: Where to look for errors

Unfixability: **risk of not being able to fix a bug.**

Language-typical errors**: such as wild pointers in C. See**

- Bruce Webster, *Pitfalls of Object-Oriented Development*
- Michael Daconta et al. *Java Pitfalls*

Criticality**: severity of failure of very important features.**

Popularity**: likelihood or consequence if much used features fail.**

Market: **severity of failure of key differentiating features.**

Bad publicity: **a bug may appear in PC Week.**

Liability: **being sued.**

Adapted from James Bach's lecture notes

Risks: Where to look for errors

Quality Categories:

- Capability
- Reliability
- Usability
- Performance
- Installability
- Compatibility – Efficiency
- Supportability – Localizability
- Testability – Portability – Maintainability

Each quality category is a risk category, as in: "the risk of unreliability."

Derived from James Bach's Satisfice Model

Bug Patterns as a Source of Risk

***Testing Computer Software* lays out a set of 480 common defects. You can use these or develop your own list.**

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

For an up to date analysis along these lines, see Giri Vijayaraghavan's paper in this conference.

Build Your Own Model of Bug Patterns

Too many people start and end with the TCS bug list. It is 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:

- 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.
- Client 1 and 2 both want to enter a name and client 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 are accepted, and we now have two instances of the same name.
- As we see these, we develop a library of issues. The discovery method is exploratory, requires sophistication with the underlying technology.
- Capture winning themes for testing in charts or in scripts-on-their-way to being automated.

Risk-Based Testing

Exercises

- Given a list of ways that the program *could* fail, for each case:
  - » Describe two ways to test for that possible failure
  - » Explain how to make your tests more powerful against that type of possible failure
  - » Explain why your test is powerful against that hazard.
- Given a list of test cases
  - » Identify a hazard that the test case might have power against
  - » Explain why this test is powerful against that hazard.

Risk-Based Testing Exercises

Take a function in the software under test. Make a hypothetical change to the product. Do a risk analysis:

- For N (10 to 30) minutes, the students privately list risk factors and associated tests (ask for 2 tests per factor)
- Break
- Class brainstorm onto flipcharts--list risk factors.
- Pairs project--fill out the charts:
  - » Students work in pairs to complete their chart
  - » Each chart has one risk factor as a title
  - » List tests derived from risk (factor) on body of page
  - » When a pair is done with its page, they move around the room, adding notes to other pairs' pages.
- Paired testing, one chart per pair, test SUT per that risk factor.

Risk-Based Testing Exercises

Collect or create some test cases for the software under test. Make a variety of tests:

- Mainstream tests that use the program in "safe" ways
- Boundary tests
- Scenario tests
- Wandering walks through the program
- Etc.
- If possible, use tests the students have suggested previously.

For each test, ask:

- How will this test find a defect?
- What kind of defect did the test author probably have in mind?
- What power does this test have against that kind of defect? Is there a more powerful test? A more powerful suite of tests?

Risk-Based Test Management

Project risk management involves

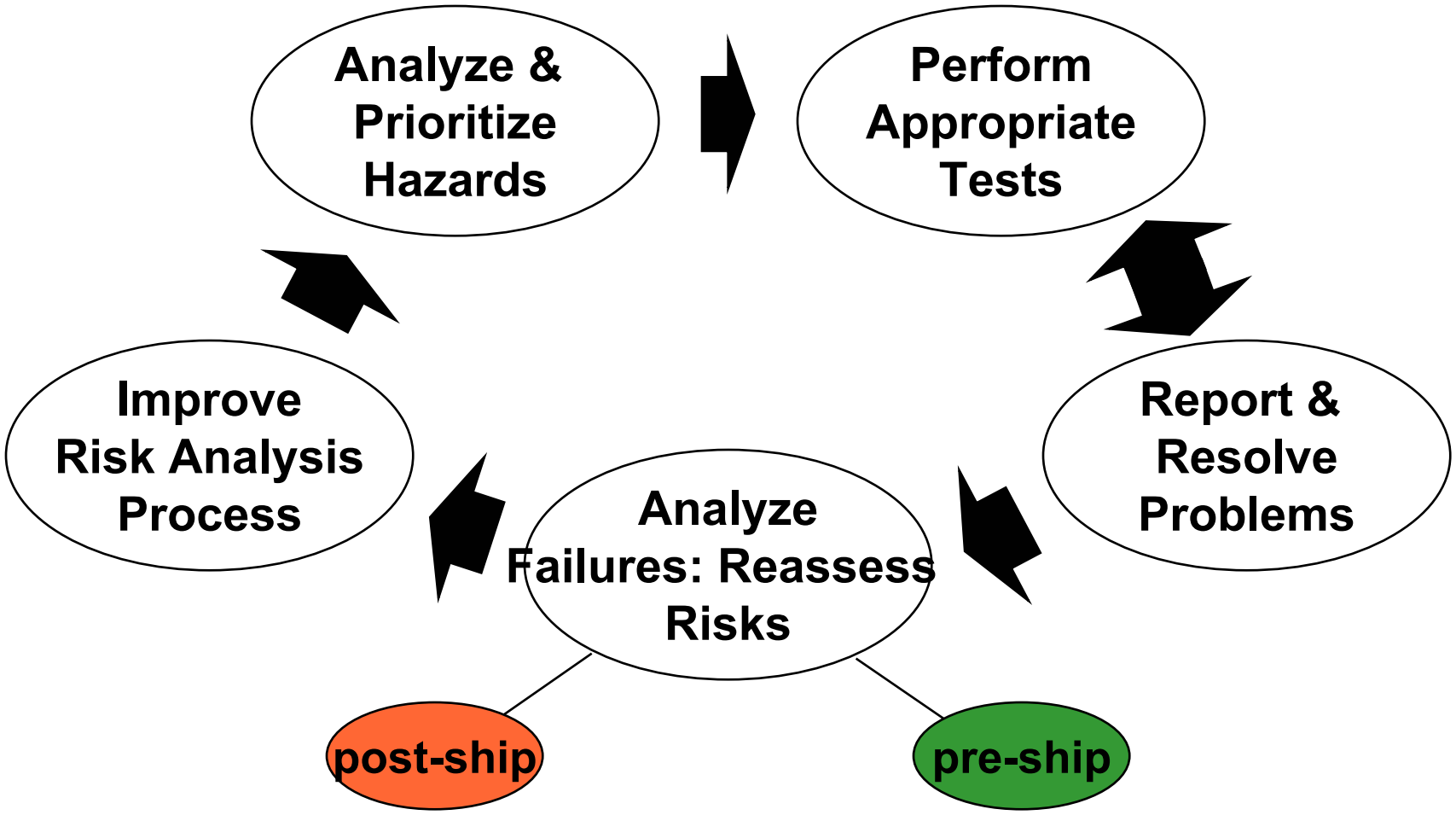
- Identification of the different risks to the project (issues that might cause the project to fail or to fall behind schedule or to cost too much or to dissatisfy customers or other stakeholders)
- Analysis of the potential costs associated with each risk
- Development of plans and actions to reduce the likelihood of the risk or the magnitude of the harm
- Continuous assessment or monitoring of the risks (or the actions taken to manage them)

Useful material available free at http://seir.sei.cmu.edu

http://www.coyotevalley.com (Brian Lawrence)

Good paper by Stale Amland, *Risk Based Testing and Metrics,* 16th International Conference on Testing Computer Software, 1999.

Risk-Driven Testing Cycle



Categories of Risk Sources

- Mission and goals
- Decision drivers
- Organization management
- Customer / end user
- Budget / cost
- Schedule
- Project characteristics
- Development process
- Development environment
- Personnel
- Operational environment
- New technology

Project Consequences

- Cost overruns
- Schedule slips
- Inadequate functionality
- Canceled projects
- Sudden personnel changes
- Customer dissatisfaction
- Loss of company image
- Demoralized staff
- Poor product performance
- Legal proceedings

Risk-Based Test Management

Tasks

- List all areas of the program that could require testing
- On a scale of 1-5, assign a probability-of-failure estimate to each
- On a scale of 1-5, assign a severity-of-failure estimate to each
- For each area, identify the specific ways that the program might fail and assign probability-of-failure and severity-of-failure estimates for those
- Prioritize based on estimated risk
- Develop a stop-loss strategy for testing untested or lightly-tested areas, to check whether there is easy-to-find evidence that the areas estimated as low risk are not actually low risk.

Risk-Based Testing: Some Papers of Interest

- Stale Amland, Risk Based Testing
- James Bach, Reframing Requirements Analysis
- James Bach, Risk and Requirements- Based Testing
- James Bach, James Bach on Risk-Based Testing
- Stale Amland & Hans Schaefer, Risk based testing, a response
- Carl Popper, Conjectures & Refutations

Paradigms:

Regression Testing

Automating Regression Testing

This is the most commonly discussed automation approach:

- create a test case
- run it and inspect the output
- if the program fails, report a bug and try again later
- if the program passes the test, save the resulting outputs
- in future tests, run the program and compare the output to the saved results. Report an exception whenever the current output and the saved output don't match.

Potential Regression Advantages

- Dominant paradigm for automated testing.
- Straightforward
- Same approach for all tests
- Relatively fast implementation
- Variations may be easy
- Repeatable tests

The GUI Regression Automation Problem

Prone to failure because of difficult financing, architectural, and maintenance issues.

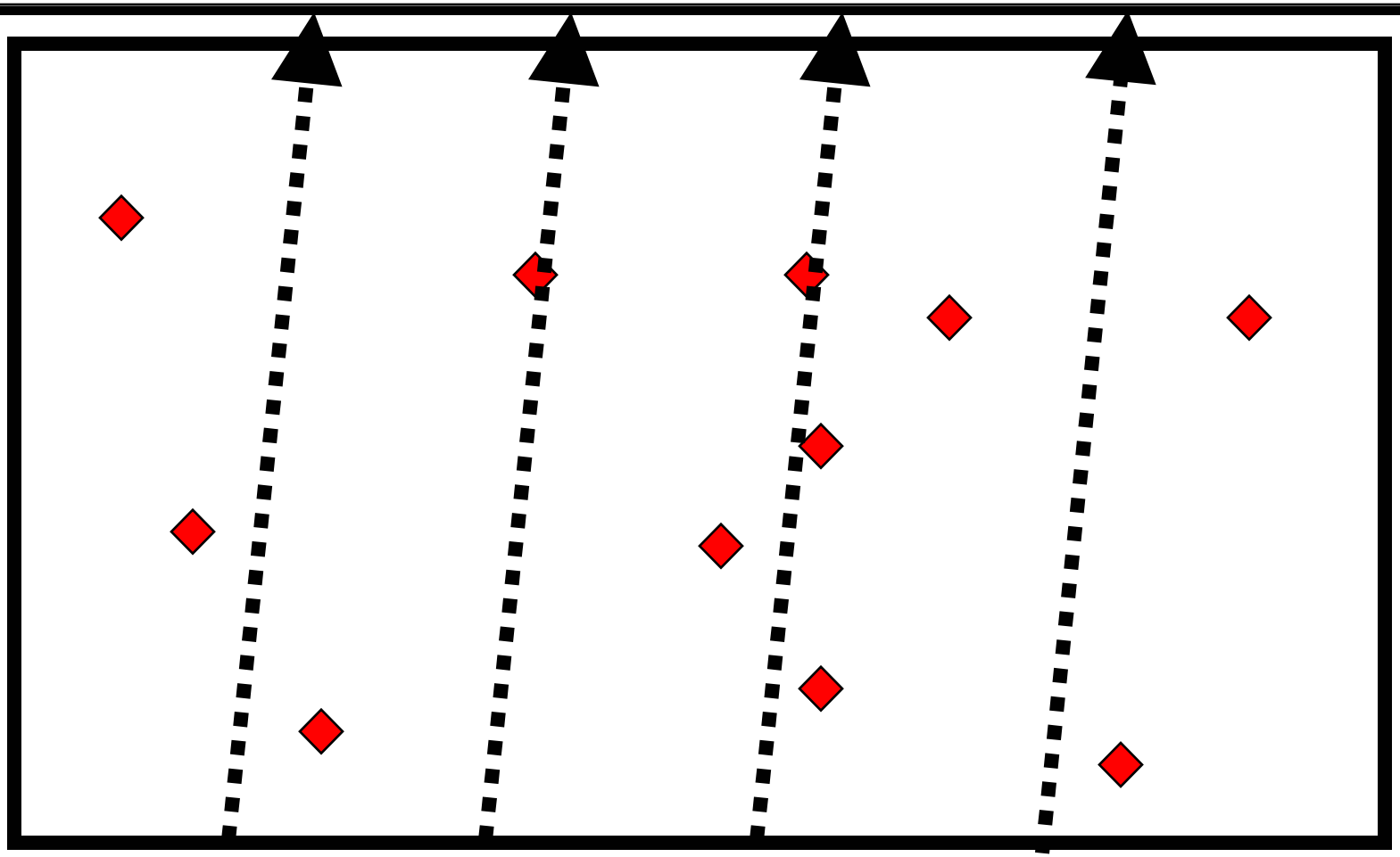
Low power (in its traditional form) even if successful.

Extremely valuable under some circumstances.

THERE ARE MANY ALTERNATIVES THAT CAN BE MORE APPROPRIATE UNDER OTHER CIRCUMSTANCES.

If your only tool is a hammer, everything looks like a nail.

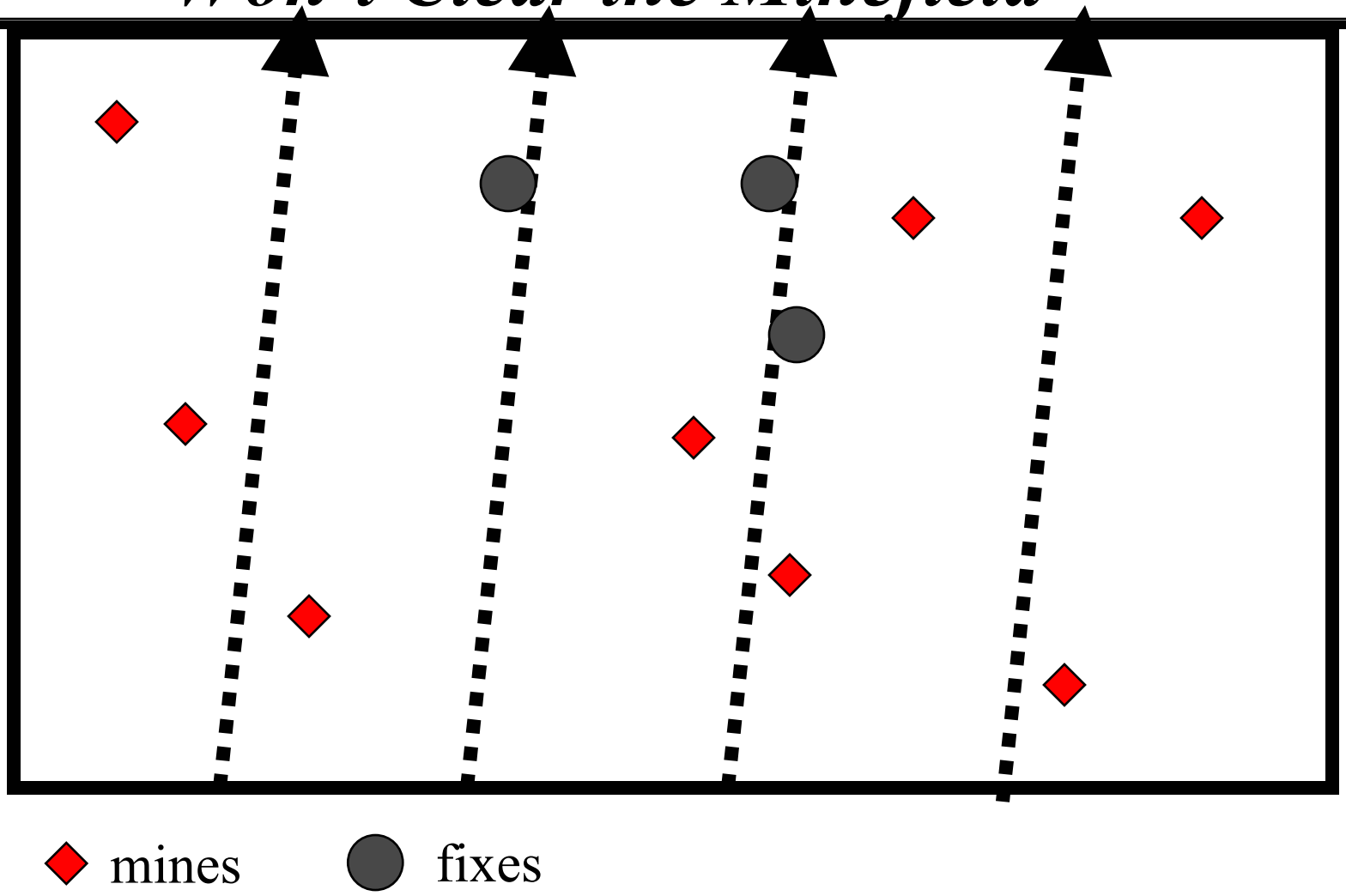
Testing Analogy: Clearing Mines



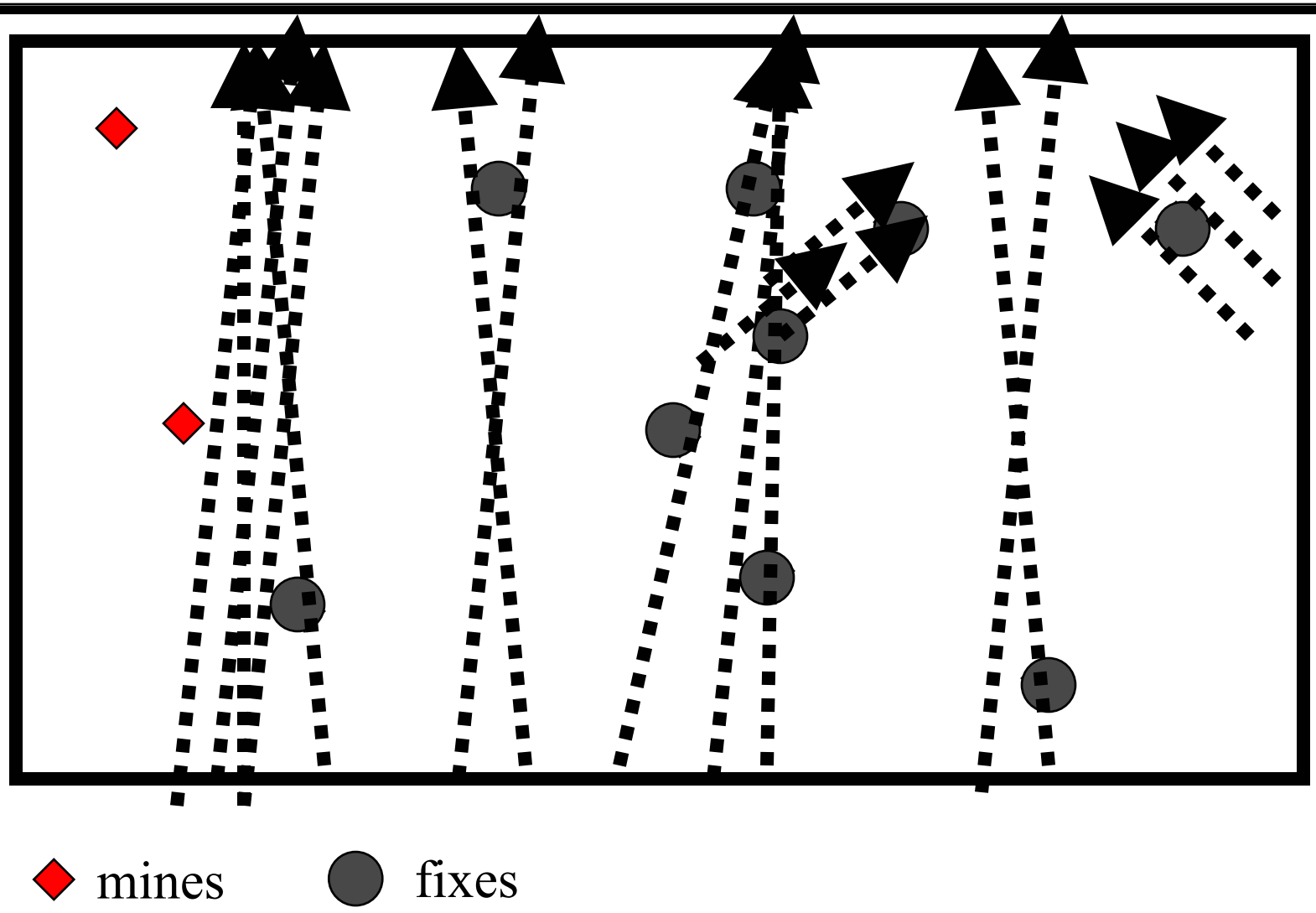
mines

This analogy was first presented by Brian Marick.
These slides are from James Bach..

Totally Repeatable Tests Won't Clear the Minefield



Variable Tests are Often More Effective



GUI Regression Strategies: Some Papers of Interest

- **Chris Agruss, Automating Software Installation Testing**
- **James Bach, Test Automation Snake Oil**
- **Hans Buwalda, Testing Using Action Words**
- **Hans Buwalda, Automated testing with Action Words: Abandoning Record & Playback**
- **Elisabeth Hendrickson, The Difference between Test Automation Failure and Success**
- **Cem Kaner, Avoiding Shelfware: A Manager's View of Automated GUI Testing**
- **John Kent, Advanced Automated Testing Architectures**
- **Bret Pettichord, Success with Test Automation**
- **Bret Pettichord, Seven Steps to Test Automation Success**
- **Keith Zambelich, Totally Data-Driven Automated Testing**

Exploratory Testing

Many of the ideas in these notes were reviewed and extended by my colleagues at the 7th Los Altos Workshop on Software Testing. I appreciate the assistance of the other LAWST 7 attendees: Brian Lawrence, III, Jack Falk, Drew Pritsker, Jim Bampos, Bob Johnson, Doug Hoffman, Chris Agruss, Dave Gelperin, Melora Svoboda, Jeff Payne, James Tierney, Hung Nguyen, Harry Robinson, Elisabeth Hendrickson, Noel Nyman, Bret Pettichord, & Rodney Wilson.

Doing Exploratory Testing

- Keep your mission clearly in mind.
- Distinguish between testing and observation.
- While testing, be aware of the limits of your ability to detect problems.
- Keep notes that help you report what you did, why you did it, and support your assessment of product quality.
- Keep track of questions and issues raised in your exploration.

Problems to be Wary of…

- **Habituation may cause you to miss problems.**
- **Lack of information may impair exploration.**
- **Expensive or difficult product setup may increase the cost of exploring.**
- **Exploratory feedback loop my be too slow.**
- **Old problems may pop up again and again.**
- **High MTBF may not be achievable without well defined test cases and procedures, in addition to exploratory approach.**

Styles of Exploration

Experienced, skilled explorers develop their own styles.

When you watch or read different skilled explorers, you see very different approaches. This is a survey of the approaches that I've seen.

Characterizing the Styles

At the heart of all of the approaches to exploratory testing (all of the styles), I think we find *questions* and *questioning skills*. As you consider the styles that follow, try to characterize them with respect to each other:

- All of the approaches are methodical, but do they focus on the
  - » Method of questioning?
  - » Method of describing or analyzing the product?
  - » The details of the product?
  - » The patterns of use of the product?
  - » The environment in which the product is run?
- To what extent would this style benefit from group interaction?

Characterizing the Styles

- What skills and knowledge does the style require or assume?
  - » Programming / debugging
  - » Knowledge of applications of this type and how they fail
  - » Knowledge of the use of applications of this type
  - » Deep knowledge of the software under test
  - » Knowledge of the system components (h/w or s/w or network) that are the context for the application
  - » Long experience with software development projects and their typical problems
  - » Requirements analysis or problem decomposition techniques
  - » Mathematics, probability, formal modeling techniques

Query: Are any of these techniques appropriate to novices? Can we train novices in exploration?

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Styles of Exploration

- Basic Hunches
  - » **"Random"**
  - » **Similarity to previous errors**
  - » **Following up gossip and predictions**
  - » **Follow up recent changes**
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Random

- People who don't understand exploratory testing describe it as "random testing." They use phrases like "random tests", "monkey tests", "dumb user tests". This is probably the most common characterization of exploratory testing.
- This describes very little of the type of testing actually done by skilled exploratory testers.

Similarity to Previous Errors

James Bach once described exploratory testers as

mental pack rats who horde memories of every bug they've ever seen.

The way they come up with cool new tests is by analogy:

Gee, I saw a program kind of like this before, and it had a bug like this.

How could I test this program to see if it has the same old bug?

A more formal variation:

- Create a potential bugs list, like the Appendix A of *Testing Computer Software*

Another related type of analogy:

- Sample from another product's test docs.

Follow Up Gossip And Predictions

Sources of gossip:

- directly from programmers, about their own progress or about the progress / pain of their colleages
- from attending code reviews (for example, at some reviews, the question is specifically asked in each review meeting, "What do you think is the biggest risk in this code?")
- from other testers, writers, marketers, etc.

Sources of predictions

- notes in specs, design documents, etc. that predict problems
- predictions based on the current programmer's history of certain types of defects

Follow Up Recent Changes

Given a current change

- tests of the feature / change itself
- tests of features that interact with this one
- tests of data that are related to this feature or data set
- tests of scenarios that use this feature in complex ways

Styles of Exploration

- Hunches
- **Models**
  - » **Failure models**
  - » **Architecture diagrams**
  - » **Bubble diagrams**
  - » **Data relationships**
  - » **Procedural relationships**
  - » **Model-based testing (state matrix)**
  - » **Requirements definition**
  - » **Functional relationships (for regression testing)**
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Failure Model: Whittaker & Jorgenson: "The fundamental cause of software errors"

Constraint violations

- input constraints
  - » such as buffer overflows
- output constraints
- computation
  - » look for divide by zeros and rounding errors. Figure out inputs that you give the system that will make it not recognize the wrong outputs.
- data violations
- Really good for finding security holes

Models and Exploration

We usually think of modeling in terms of preparation for formal testing, but there is no conflict between modeling and exploration. Both types of tests start from models. The difference is that in exploratory testing, our emphasis is on execution (try it *now*) and learning from the results of execution rather than on documentation and preparation for later execution.

Architecture Diagrams

Work from a high level design (map) of the system

- pay primary attention to interfaces between components or groups of components. We're looking for cracks that things might have slipped through
- what can we do to screw things up as we trace the flow of data or the progress of a task through the system?

You can build the map in an architectural walkthrough

- Invite several programmers and testers to a meeting. Present the programmers with use cases and have them draw a diagram showing the main components and the communication among them. For a while, the diagram will change significantly with each example. After a few hours, it will stabilize.
- Take a picture of the diagram, blow it up, laminate it, and you can use dry erase markers to sketch your current focus.
- Planning of testing from this diagram is often done jointly by several testers who understand different parts of the system.

Bubble (Reverse State) Diagrams

To troubleshoot a bug, a programmer will often work the code backwards, starting with the failure state and reading for the states that could have led to it (and the states that could have led to those).

The tester imagines a failure instead, and asks how to produce it.

- Imagine the program being in a failure state. Draw a bubble.
- What would have to have happened to get the program here? Draw a bubble for each immediate precursor and connect the bubbles to the target state.
- For each precursor bubble, what would have happened to get the program there? Draw more bubbles.
- More bubbles, etc.
- Now trace through the paths and see what you can do to force the program down one of them.

Bubble (Reverse State) Diagrams

Example:

- *How could we produce a paper jam (as a result of defective firmware, rather than as a result of jamming the paper?) The laser printer feeds a page of paper at a steady pace. Suppose that after feeding, the system reads a sensor to see if there is anything left in the paper path. A failure would result if something was wrong with the hardware or software controlling or interpreting the paper feeding (rollers, choice of paper origin, paper tray), paper size, clock, or sensor.*

Data Relationships

- Pick a data item
- Trace its flow through the system
- What other data items does it interact with?
- What functions use it?
- Look for inconvenient values for other data items or for the functions, look for ways to interfere with the function using this data item

Data Relationship Chart

Field	*Entry Source*	*Display*	*Print*	*Related Variable*	*Relationship*
Variable 1	*Any way you can change values in V1*	*After V1 & V2 are brought to incompatible values, what are all the ways to display them?*	*After V1 & V2 are brought to incompatible values, what are all the ways to display or use them?*	Variable 2	Constraint to a range
Variable 2	*Any way you can change values in V1*			Variable 1	Constraint to a range

Procedural Relationships

- Pick a task
- Step by step, describe how it is done and how it is handled in the system (to as much detail as you know)
- Now look for ways to interfere with it, look for data values that will push it toward other paths, look for other tasks that will compete with this one, etc.

Improvisational Testing

The originating model here is of the test effort, not (explicitly) of the software.

Another approach to ad hoc testing is to treat it as improvisation on a theme, not unlike jazz improvisation in the musical world. For example, testers often start with a Test Design that systematically walks through all the cases to be covered. Similarly, jazz musicians often start with a musical score or "lead sheet" for the tunes on which they intend to improvise.

In this version of the ad hoc approach, the tester is encouraged to take off on tangents from the original Test Design whenever it seems worthwhile. In other words, the tester uses the test design but invents variations. This approach combines the strengths of both structured and unstructured testing: the feature is tested as specified in the test design, but several variations and tangents are also tested. On this basis, we expect that the improvisational approach will yield improved coverage.

Improvisational Testing

The originating model here is of the test effort, not (explicitly) of the software.

Improvisational techniques are also useful when verifying that defects have been fixed. Rather than simply verifying that the steps to reproduce the defect no longer result in the error, the improvisational tester can test more deeply "around" the fix, ensuring that the fix is robust in a more general sense.

Johnson & Agruss, Ad Hoc Software Testing:

Exploring the Controversy of Unstructured Testing

STAR'98 WEST

Styles of Exploration

- Hunches
- Models
- **Examples**
  - » **Use cases**
  - » **Simple walkthroughs**
  - » **Positive testing**
  - » **Scenarios**
  - » **Soap operas**
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Use Cases

- List the users of the system
- For each user, think through the tasks they want to do
- Create test cases to reflect their simple and complex uses of the system

Simple Walkthroughs

Test the program broadly, but not deeply.

- Walk through the program, step by step, feature by feature.
- Look at what's there.
- Feed the program simple, nonthreatening inputs.
- Watch the flow of control, the displays, etc.

Positive Testing

- Try to get the program working in the way that the programmers intended it.
- One of the points of this testing is that you educate yourself about the program. You are looking at it and learning about it from a sympathetic viewpoint, using it in a way that will show you what the value of the program is.
- This is true “positive” testing—you are trying to make the program show itself off, not just trying to confirm that all the features and functions are there and kind of sort of working.

Scenarios

The ideal scenario has several characteristics:

- It is realistic (e.g. it comes from actual customer or competitor situations).
- There is no ambiguity about whether a test passed or failed.
- The test is complex, that is, it uses several features and functions.
- There is a stakeholder who will make a fuss if the program doesn't pass this scenario.

For more on scenarios, see the scenarios paradigm discussion.

Styles of Exploration

- Hunches
- Models
- Examples
- **Invariances**
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Invariances

These are tests run by making changes that shouldn't affect the program. Examples:

- load fonts into a printer in different orders
- set up a page by sending text to the printer and then the drawn objects or by sending the drawn objects and then the text
- use a large file, in a program that should be able to handle any size input file (and see if the program processes it in the same way)
- mathematical operations in different but equivalent orders

==========

John Musa — Intro to his book, *Reliable Software Engineering*, says that you should use different values within an equivalance class. For example, if you are testing a flight reservation system for two US cities, vary the cities. They shouldn't matter, but sometimes they do.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- **Interference**
  - » **Interrupt**
  - » **Change**
  - » **Stop**
  - » **Pause**
  - » **Swap**
  - » **Compete**
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Interference Testing

We're often looking at asynchronous events here. One task is underway, and we do something to interfere with it.

In many cases, the critical event is extremely time sensitive. For example:

- An event reaches a process just as, just before, or just after it is timing out or just as (before / during / after) another process that communicates with it will time out listening to this process for a response. ("Just as?"—if special code is executed in order to accomplish the handling of the timeout, "just as" means during execution of that code)
- An event reaches a process just as, just before, or just after it is servicing some other event.
- An event reaches a process just as, just before, or just after a resource needed to accomplish servicing the event becomes available or unavailable.

Interrupt

Generate interrupts

- from a device related to the task (e.g. pull out a paper tray, perhaps one that isn't in use while the printer is printing)
- from a device unrelated to the task (e.g. move the mouse and click while the printer is printing)
- from a software event

Change

Change something that this task depends on

- swap out a floppy
- change the contents of a file that this program is reading
- change the printer that the program will print to (without signaling a new driver)
- change the video resolution

Stop

- Cancel the task (at different points during its completion)
- Cancel some other task while this task is running
  - » a task that is in communication with this task (the core task being studied)
  - » a task that will eventually have to complete as a prerequisite to completion of this task
  - » a task that is totally unrelated to this task

Pause

- Find some way to create a temporary interruption in the task.
- Pause the task
  - » for a short time
  - » for a long time (long enough for a timeout, if one will arise)
- Put the printer on local
- Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

Swap (out of memory)

- Swap the process out of memory while it is running (e.g. change focus to another application and keep loading or adding applications until the application under test is paged to disk.
  - » Leave it swapped out for 10 minutes or whatever the timeout period is. Does it come back? What is its state? What is the state of processes that are supposed to interact with it?
  - » Leave it swapped out *much* longer than the timeout period. Can you get it to the point where it is supposed to time out, then send a message that is supposed to be received by the swapped-out process, then time out on the time allocated for the message? What are the resulting state of this process and the one(s) that tried to communicate with it?
- Swap a related process out of memory while the process under test is running.

Compete

Examples:

Compete for a device (such as a printer)

- put device in use, then try to use it from software under test
- start using device, then use it from other software
- If there is a priority system for device access, use software that has higher, same and lower priority access to the device before and during attempted use by software under test

Compete for processor attention

- some other process generates an interrupt (e.g. ring into the modem, or a time-alarm in your contact manager)
- try to do something during heavy disk access by another process

Send this process another job while one is underway

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- **Error Handling**
- Troubleshooting
- Group Insights
- Specifications

Error Handling

The usual suspects:

- Walk through the error list.
  - » Press the wrong keys at the error dialog.
  - » Make the error several times in a row (do the equivalent kind of probing to defect follow-up testing).
- Device-related errors (like disk full, printer not ready, etc.)
- Data-input errors (corrupt file, missing data, wrong data)
- Stress / volume (huge files, too many files, tasks, devices, fields, records, etc.)

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- **Troubleshooting**
- Group Insights
- Specifications

Troubleshooting

We often do exploratory tests when we troubleshoot bugs:

- Bug analysis:
  - » simplify the bug by deleting or simplifying steps
  - » simplify the bug by simplifying the configuration (or the tools in the background)
  - » clarify the bug by running variations to see what the problem is
  - » clarify the bug by identifying the version that it entered the product
  - » strengthen the bug with follow-up tests (using repetition, related tests, related data, etc.) to see if the bug left a side effect
  - » strengthen the bug with tests under a harsher configuration
- Bug regression: vary the steps in the bug report when checking if the bug was fixed

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- **Group Insights**
  - » **Brainstormed test lists**
  - » **Group discussion of related components**
  - » **Fishbone analysis**
- Specifications

Brainstormed Test Lists

We saw a simple example of this at the start of the class. You brainstormed a list of tests for the two-variable, two-digit problem:

- The group listed a series of cases (test case, why)
- You then examined each case and the class of tests it belonged to, looking for a more powerful variation of the same test.
- You then ran these tests.

You can apply this approach productively to any part of the system.

Group Discussion of Related Components

The objective is to test the interaction of two or more parts of the system.

The people in the group are very familiar with one or more of parts. Often, no one person is familiar with all of the parts of interest, but collectively the ideal group knows all of them.

The group looks for data values, timing issues, sequence issues, competing tasks, etc. that might screw up the orderly interaction of the components under study.

Fishbone Analysis

- Fishbone analysis is a traditional failure analysis technique. Given that the system has shown a specific failure, you work backwards through precursor states (the various paths that could conceivably lead to this observed failure state).
- As you walk through, you say that Event A couldn't have happened unless Event B or Event C happened. And B couldn't have happened unless B1 or B2 happened. And B1 couldn't have happened unless X happened, etc.
- While you draw the chart, you look for ways to prove that X (whatever, a precursor state) *could* actually have been reached. If you succeed, you have found one path to the observed failure.
- As an exploratory test tool, you use "risks" instead of failures. You imagine a possible failure, then walk backwards asking if there is a way to achieve it. You do this as a group, often with a computer active so that you can try to get to the states as you go.

Paired Exploratory Testing--Acknowledgment

These notes are from a presentation by Cem Kaner & James Bach.

The following, paired testing, slides developed out of several projects.

We (Bach, Kaner) particularly acknowledge the help and data from participants in the First and Second Workshops on Heuristic and Exploratory Techniques (Front Royal, VA, November 2000 and March 2001, hosted by James Bach and facilitated by Cem Kaner), those being Jon Bach, Stephen Bell, Rex Black, Robyn Brilliant, Scott Chase, Sam Guckenheimer, Elisabeth Hendrickson, Alan A. Jorgensen, Brian Lawrence, Brian Marick, Mike Marduke, Brian McGrath, Erik Petersen, Brett Pettichord, Shari Lawrence Pfleeger, Becky Winant, and Ron Wilson.

Additionally, we thank Noel Nyman and Ross Collard for insights and James Whittaker for co-hosting one of the two paired testing trials at Florida Tech.

A testing pattern on paired testing was drafted by Brian Marick, based on discussions at the Workshop on Patterns of Software Testing (POST 1) in Boston, January 2001 (hosted primarily by Sam Guckenheimer / Rational and Brian Marick, facilitated by Marick). The latest draft is at "Pair Testing" pattern) (<http://www.testing.com/test-patterns/patterns/pair-testing.pdf>).

Paired Exploratory Testing

- **Based on our (and others') observations of effective testing workgroups at several companies. We noticed several instances of high productivity, high creativity work that involved testers grouping together to analyze a product or to scheme through a test or to run a series of tests. We also saw/used it as an effective training technique.**
- **In 2000, we started trying this out, at WHET, at Satisfice, and at one of Satisfice's clients. The results were spectacular. We obtained impressive results in quick runs at Florida Tech as well, and have since received good reports from several other testers.**

Paired Programming

- Developed independently of paired testing, but many of the same problems and benefits apply.
- The eXtreme Programming community has a great deal of experience with paired work, much more than we do, offers many lessons:
  - » Kent Beck, *Extreme Programming Explained*
  - » Ron Jeffries, Ann Anderson & Chet Hendrickson, *Extreme Programming Installed*
- Laurie Williams of NCSU does research in pair programming. For her publications, see <http://collaboration.csc.ncsu.edu/laurie/>

What is Paired Testing

- Two testers and (typically) one machine.
- Typically (as in XP)
  - » Pairs work together voluntarily. One person might pair with several others during a day.
  - » A given testing task is the responsibility of one person, who recruits one or more partners (one at a time) to help out.
- We've seen stable pairs who've worked together for years.
- One tester strokes the keys (but the keyboard may pass back and forth in a session) while the other suggests ideas or tests, pays attention and takes notes, listens, asks questions, grabs reference material, etc.

A Paired Testing Session

Start with a charter

- Testers might operate from a detailed project outline, pick a task that will take a day or less
- Might (instead or also) create a flipchart page that outlines this session's work or the work for the next few sessions.
  - » An exploratory testing session lasts about 60-90 minutes.
- The charter for a session might include what to test, what tools to use, what testing tactics to use, what risks are involved, what bugs to look for, what documents to examine, what outputs are desired, etc.

Benefits of Paired Testing

- Pair testing is different from many other kinds of pair work because testing is an *idea generation activity* rather than a plan implementation activity. Testing is a heuristic search of an open-ended and multi-dimensional space.
- Pairing has the effect of forcing each tester to explain ideas and react to ideas. When one tester must phrase his thoughts to another tester, that simple process of phrasing seems to bring the ideas into better focus and naturally triggers more ideas.
- If faithfully performed, we believe this will result in more and better ideas that inform the tests.

Benefits of Paired Testing

Generate more ideas

- Naturally encourages creativity
- More information and insight available to apply to analysis of a design or to any aspect of the testing problem
- Supports the ability of one tester to stay focused and keep testing. This has a major impact on creativity.

More fun

Benefits of Paired Testing

Helps the tester stay on task. Especially helps the tester pursue a streak of insight (an exploratory vector).

- A flash of insight need not be interrupted by breaks for note-taking, bug reporting, and follow-up replicating. The non-keyboard tester can:
  - » Keep key notes while the other follows the train of thought
  - » Try to replicate something on a second machine
  - » Grab a manual, other documentation, a tool, make a phone call, grab a programmer--get support material that the other tester needs.
  - » Record interesting candidates for digression

Also, the fact that two are working together limits the willingness of others to interrupt them, especially with administrivia.

Benefits of Paired Testing

Better Bug Reporting

- Better reproducibility
- Everything reported is reviewed by a second person.
- Sanity/reasonability check for every design issue
  - » (example from Kaner/Black on Star Office tests)

Great training

- Good training for novices
- Keep learning by testing with others
- Useful for experienced testers when they are in a new domain

Benefits of Paired Testing

Additional technical benefits

- Concurrency testing is facilitated by pairs working with two (or more) machines.
- Manual load testing is easier with several people.
- When there is a difficult technical issue with part of the project, bring in a more knowledgeable person as a pair

Risks and Suggestions

- Paired testing is *not* a vehicle for fobbing off errand-running on a junior tester. The pairs are partners, the junior tester is often the one at the keyboard, and she is always allowed to try out her own ideas.
- Accountability must belong to one person. Beck and Jeffries, et al. discuss this in useful detail. One member of the pair owns the responsibility for getting the task done.
- Some people are introverts. They need time to work alone and recharge themselves for group interaction.
- Some people have strong opinions and don't work well with others. Coaching may be essential.

Risks and Suggestions

Have a coach available.

- Generally helpful for training in paired testing and in the conduct of any type of testing
- When there are strong personalities at work, a coach can help them understand their shared and separate responsibilities and how to work effectively together.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insight
- **Specifications**
  - » **Active reading -- Ambiguity analysis**
  - » **User manual**
  - » **Consistency heuristics**

Active Reading (Ambiguity Analysis)

There are all sorts of sources of ambiguity in software design and development.

- In the wording or interpretation of specifications or standards
- In the expected response of the program to invalid or unusual input
- In the behavior of undocumented features
- In the conduct and standards of regulators / auditors
- In the customers' interpretation of their needs and the needs of the users they represent
- In the definitions of compatibility among 3rd party products

Whenever there is ambiguity, there is a strong opportunity for a defect (at least in the eyes of anyone who understands the world differently from the implementation).

One interesting workbook: Cecile Spector, *Saying One Thing, Meaning Another*.

User Manual

Write part of the user manual and check the program against it as you go. Any writer will discover bugs this way. An exploratory tester will discover quite a few this way.

Consistency Heuristics:

- Consistent with History: **Present function behavior is consistent with past behavior.**
- Consistent with an Image: **Function behavior is consistent with an image that the organization wants to project.**
- Consistent with Comparable Products: **Function behavior is consistent with that of similar functions in comparable products.**
- Consistent with Claims: **Function behavior is consistent with what people say it's supposed to be.**
- Consistent with User Values: **Function behavior is consistent with what we think users want.**
- Consistent within Product: **Function behavior is consistent with behavior of comparable functions or functional patterns within the product.**
- Consistent with Purpose: **Function behavior is consistent with its apparent purpose.**

Exploratory Testing: Some Papers of Interest

- Chris Agruss & Bob Johnson, *Ad Hoc Software Testing Exploring the Controversy of Unstructured Testing*
- Cem Kaner & James Bach, *Exploratory Testing. (Available later in the materials)*
- Whittaker, *How to Break Software*

Paradigms:

Scenario Testing

Scenarios

Some ways to trigger thinking about scenarios:

- **Benefits-driven:** People want to achieve X. How will they do it, for the following X's?
- **Sequence-driven:** People (or the system) typically does task X in an order. What are the most common orders (sequences) of subtasks in achieving X?
- **Transaction-driven:** We are trying to complete a specific transaction, such as opening a bank account or sending a message. What are all the steps, data items, outputs and displays, etc.?
- **Get use ideas from competing product:** Their docs, advertisements, help, etc., all suggest best or most interesting uses of their products. How would our product do these things?

Scenarios

Some ways to trigger thinking about scenarios:

- **Competitor's output driven**: Hey, look at these cool documents they can make. Look (think of Netscape's superb handling of often screwy HTML code) at how well they display things. How do we do with these?
- **Customer's forms driven**: Here are the forms the customer produces in her business. How can we work with (read, fill out, display, verify, whatever) them?

Scenarios

There are several related traditions or approaches:

- Use cases and scenarios within use case analysis (a scenario is an instantiation of a use case).
- Customer stories within extreme programming
- Usage scenarios derived from human factors, training, or documentation development task-analysis
- Hans Buwalda's soap operas

Soap Operas

- Build a scenario based on real-life experience. This means client/customer experience.
- Exaggerate each aspect of it:
  - » example, for each variable, substitute a more extreme value
  - » example, if a scenario can include a repeating element, repeat it lots of times
  - » make the environment less hospitable to the case (increase or decrease memory, printer resolution, video resolution, etc.)
- Create a real-life story that combines all of the elements into a test case narrative.

(Thanks to Hans Buwalda for developing this approach and patiently explaining it to me.)

Soap Operas

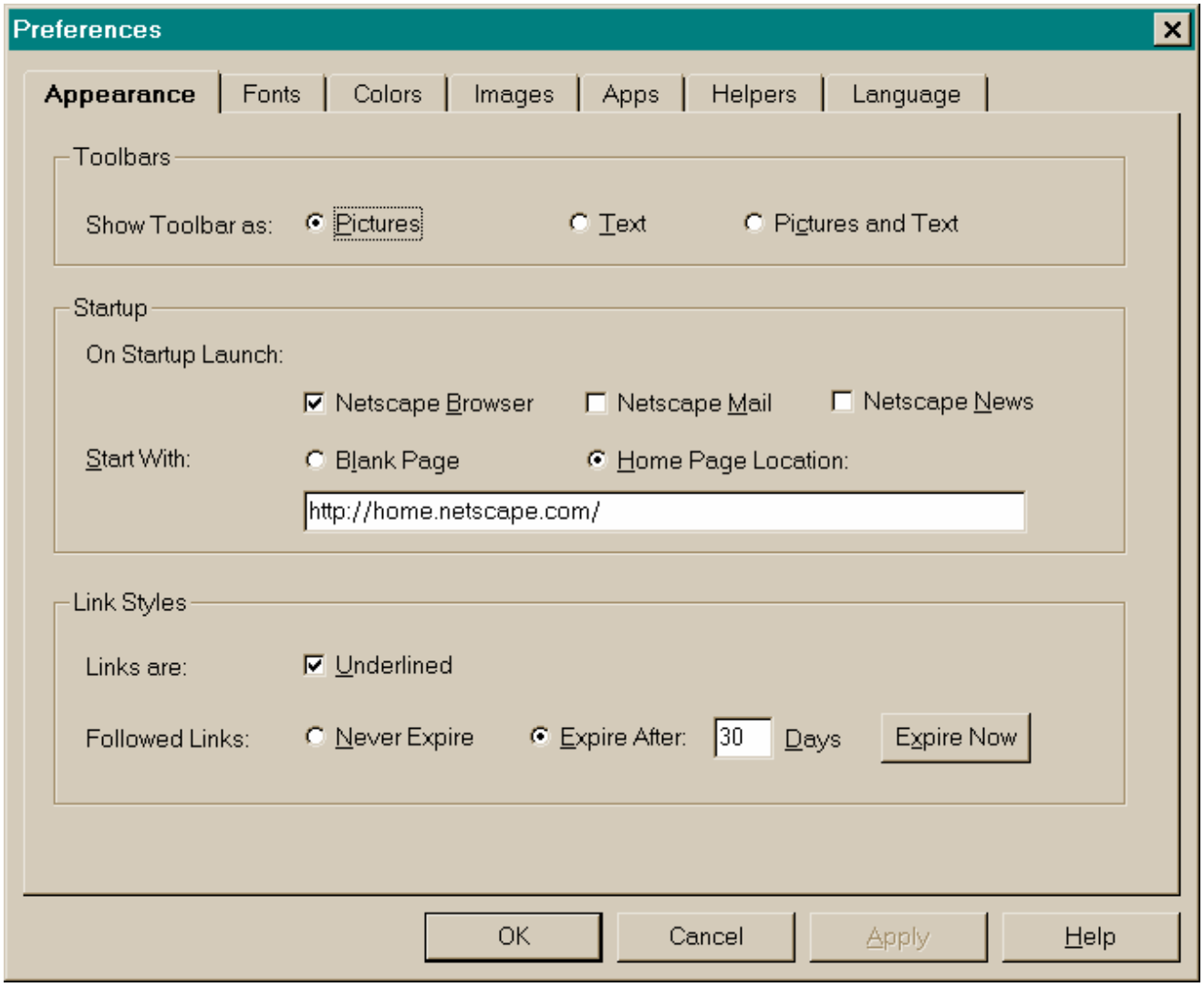
(As these have evolved, Hans distinguishes between *normal soap operas*, which combine many issues based on user requirements–typically derived from meetings with the user community and probably don't exaggerate beyond normal use—and *killer soap operas*, which combine *and exaggerate to produce extreme cases*.)

Combination Testing

Combination Chart

	Var 1	Var 2	Var 3	Var 4	Var 5
Test 1	Value 11	Value 12	Value 13	Value 14	Value 15
Test 2	Value 21	Value 22	Value 23	Value 24	Value 25
Test 3	Value 31	Value 32	Value 33	Value 34	Value 35
Test 4	Value 41	Value 42	Value 43	Value 44	Value 45
Test 5	Value 51	Value 52	Value 53	Value 54	Value 55
Test 6	Value 61	Value 62	Value 63	Value 64	Value 65

Testing Variables in Combination



The Netscape Preferences dialog.

Testing Variables in Combination

If we just look at the Appearance tab of the Netscape Preferences dialog, we see the following variables:

- Toolbars -- 3 choices (P, T, B)

  (***pictures, text*** or ***both***)
- On Startup Launch -- 3 choices (B, M, N)

  (***browser***, ***mail, news***). Each of these is an independent binary choice.
- Start With -- 3 choices (B,V,E)

  (***blank*** page, home page names a ***valid*** file, home page name has a ***syntax error***)

  (Many more cases are possible, but let's keep this simple and ignore that for a few slides)
- Links -- 2 choices (D,U)

  (***don't*** underline, ***underlined***)
- Followed Links -- 2 choices (N,E)

  (***never*** expire, ***expire*** after 30 days) (Many more cases are possible)

Testing Variables in Combination

I simplified the combinations by simplifying the choices for two of the fields.

In the Start With field, I used either a valid home page name or a blank name. Some other test cases that could go into this field are:

- file name (name.htm instead of using http:// to define a protocol) on the local drive, the local network drive, or the remote drive
- maximum length file names, maximum length paths
- invalid file names and paths

For combination testing, select a few of these that look like they might interact with other variables. Test the rest independently.

Similarly for the Expire After field. This lets you enter the number of days to store links. If you use more than one value, use boundary cases, not all the numbers in the range.

In multi-variable testing, use partition analysis or other special values instead of testing all values in combination with all other variables' all values.

Testing Variables in Combination

We can create 3 x 2 x 2 x 2 x 3 x 2 x 2 = 288 different test cases by testing these variables in combination. Here are some examples, from a combination table.

	Toolbars PTB	On Startup, Browser Y/N	On Startup, Mail Y/N	On Startup, News Y/N	Start With BVE	Links DU	Followed NE
Test #1	P	Y	Y	Y	B	D	N
2	P	Y	Y	N	B	D	E
3	P	Y	N	Y	B	U	N
4	P	Y	N	N	B	U	E
5	P	Y	Y	Y	V	D	N
6	P	Y	Y	N	V	D	E
7	P	N	N	Y	V	U	N
8	P	N	N	N	V	U	E
9	P	N	Y	Y	E	D	N
10	P	N	Y	N	E	D	E
11	P	N	N	Y	E	U	N
12	P	N	N	N	E	U	E

Here are the 288 test cases. Every value of every variable is combined with every combination of the other variables.

1	P Y Y Y B D N	P N Y Y B D N	T Y Y Y B D N	T N Y Y B D N	B Y Y Y B D N	B N Y Y B D N
2	P Y Y Y V D E	P N Y Y V D E	T Y Y Y V D E	T N Y Y V D E	B Y Y Y V D E	B N Y Y V D E
3	P Y Y Y E D N	P N Y Y E D N	T Y Y Y E D N	T N Y Y E D N	B Y Y Y E D N	B N Y Y E D N
4	P Y Y Y B U E	P N Y Y B U E	T Y Y Y B U E	T N Y Y B U E	B Y Y Y B U E	B N Y Y B U E
5	P Y Y Y V U N	P N Y Y V U N	T Y Y Y V U N	T N Y Y V U N	B Y Y Y V U N	B N Y Y V U N
6	P Y Y Y E U E	P N Y Y E U E	T Y Y Y E U E	T N Y Y E U E	B Y Y Y E U E	B N Y Y E U E
7	P Y Y N B D N	P N Y N B D N	T Y Y N B D N	T N Y N B D N	B Y Y N B D N	B N Y N B D N
8	P Y Y N V D E	P N Y N V D E	T Y Y N V D E	T N Y N V D E	B Y Y N V D E	B N Y N V D E
9	P Y Y N E D N	P N Y N E D N	T Y Y N E D N	T N Y N E D N	B Y Y N E D N	B N Y N E D N
10	P Y Y N B U E	P N Y N B U E	T Y Y N B U E	T N Y N B U E	B Y Y N B U E	B N Y N B U E
11	P Y Y N V U N	P N Y N V U N	T Y Y N V U N	T N Y N V U N	B Y Y N V U N	B N Y N V U N
12	P Y Y N E U E	P N Y N E U E	T Y Y N E U E	T N Y N E U E	B Y Y N E U E	B N Y N E U E
13	P Y Y Y B D E	P N Y Y B D E	T Y Y Y B D E	T N Y Y B D E	B Y Y Y B D E	B N Y Y B D E
14	P Y Y Y V D N	P N Y Y V D N	T Y Y Y V D N	T N Y Y V D N	B Y Y Y V D N	B N Y Y V D N
15	P Y Y Y E D E	P N Y Y E D E	T Y Y Y E D E	T N Y Y E D E	B Y Y Y E D E	B N Y Y E D E
16	P Y Y Y B U N	P N Y Y B U N	T Y Y Y B U N	T N Y Y B U N	B Y Y Y B U N	B N Y Y B U N
17	P Y Y Y V U E	P N Y Y V U E	T Y Y Y V U E	T N Y Y V U E	B Y Y Y V U E	B N Y Y V U E
18	P Y Y Y E U N	P N Y Y E U N	T Y Y Y E U N	T N Y Y E U N	B Y Y Y E U N	B N Y Y E U N
19	P Y Y N B D E	P N Y N B D E	T Y Y N B D E	T N Y N B D E	B Y Y N B D E	B N Y N B D E
20	P Y Y N V D N	P N Y N V D N	T Y Y N V D N	T N Y N V D N	B Y Y N V D N	B N Y N V D N
21	P Y Y N E D E	P N Y N E D E	T Y Y N E D E	T N Y N E D E	B Y Y N E D E	B N Y N E D E
22	P Y Y N B U N	P N Y N B U N	T Y Y N B U N	T N Y N B U N	B Y Y N B U N	B N Y N B U N
23	P Y Y N V U E	P N Y N V U E	T Y Y N V U E	T N Y N V U E	B Y Y N V U E	B N Y N V U E
24	P Y Y N E U N	P N Y N E U N	T Y Y N E U N	T N Y N E U N	B Y Y N E U N	B N Y N E U N
25	P Y N Y B D E	P N N Y B D E	T Y N Y B D E	T N N Y B D E	B Y N Y B D E	B N N Y B D E
26	P Y N Y V D N	P N N Y V D N	T Y N Y V D N	T N N Y V D N	B Y N Y V D N	B N N Y V D N
27	P Y N Y E D E	P N N Y E D E	T Y N Y E D E	T N N Y E D E	B Y N Y E D E	B N N Y E D E
28	P Y N Y B U N	P N N Y B U N	T Y N Y B U N	T N N Y B U N	B Y N Y B U N	B N N Y B U N
29	P Y N Y V U E	P N N Y V U E	T Y N Y V U E	T N N Y V U E	B Y N Y V U E	B N N Y V U E
30	P Y N Y E U N	P N N Y E U N	T Y N Y E U N	T N N Y E U N	B Y N Y E U N	B N N Y E U N
31	P Y N N B D E	P N N N B D E	T Y N N B D E	T N N N B D E	B Y N N B D E	B N N N B D E
32	P Y N N V D N	P N N N V D N	T Y N N V D N	T N N N V D N	B Y N N V D N	B N N N V D N
33	P Y N N E D E	P N N N E D E	T Y N N E D E	T N N N E D E	B Y N N E D E	B N N N E D E
34	P Y N N B U N	P N N N B U N	T Y N N B U N	T N N N B U N	B Y N N B U N	B N N N B U N
35	P Y N N V U E	P N N N V U E	T Y N N V U E	T N N N V U E	B Y N N V U E	B N N N V U E
36	P Y N N E U N	P N N N E U N	T Y N N E U N	T N N N E U N	B Y N N E U N	B N N N E U N
37	P Y N Y B D N	P N N Y B D N	T Y N Y B D N	T N N Y B D N	B Y N Y B D N	B N N Y B D N
38	P Y N Y V D E	P N N Y V D E	T Y N Y V D E	T N N Y V D E	B Y N Y V D E	B N N Y V D E
39	P Y N Y E D N	P N N Y E D N	T Y N Y E D N	T N N Y E D N	B Y N Y E D N	B N N Y E D N
40	P Y N Y B U E	P N N Y B U E	T Y N Y B U E	T N N Y B U E	B Y N Y B U E	B N N Y B U E
41	P Y N Y V U N	P N N Y V U N	T Y N Y V U N	T N N Y V U N	B Y N Y V U N	B N N Y V U N
42	P Y N Y E U E	P N N Y E U E	T Y N Y E U E	T N N Y E U E	B Y N Y E U E	B N N Y E U E
43	P Y N N B D N	P N N N B D N	T Y N N B D N	T N N N B D N	B Y N N B D N	B N N N B D N
44	P Y N N V D E	P N N N V D E	T Y N N V D E	T N N N V D E	B Y N N V D E	B N N N V D E
45	P Y N N E D N	P N N N E D N	T Y N N E D N	T N N N E D N	B Y N N E D N	B N N N E D N
46	P Y N N B U E	P N N N B U E	T Y N N B U E	T N N N B U E	B Y N N B U E	B N N N B U E
47	P Y N N V U N	P N N N V U N	T Y N N V U N	T N N N V U N	B Y N N V U N	B N N N V U N
48	P Y N N E U E	P N N N E U E	T Y N N E U E	T N N N E U E	B Y N N E U E	B N N N E U E

Testing Variables in Combination

To simplify this, many testers would test variables in pairs.

That can be useful if you understand specific relationships between the variables, but if you are doing general combination testing, then restricting your attention to pairs is less efficient and less simple than you might expect.

Look at all the pairs you'd have to test, if you tested them all, pair by pair -- 109 of them. This is better than 288, but not much.

Testing Variables in Combination

	Toolbars P/T/B	Browser Y/N	Mail Y/N	News Y/N	Start B/V/E	Links D/U	Followed N/E	TOTAL PAIRS
Toolbars 3 choices	------	6	6	6	9	6	6	39
Browser 2 choices	------	------	4	4	6	4	4	22
Mail 2 choices	------	------	------	4	6	4	4	18
News 2 choices	------	------	------	------	6	4	4	14
Start 3 choices	------	------	------	------	------	6	6	12
Links 2 choices	------	------	------	------	------	------	4	4 ------
Followed 2 choices	------	------	------	------	------	------	------	***109***

Testing Variables in Combination

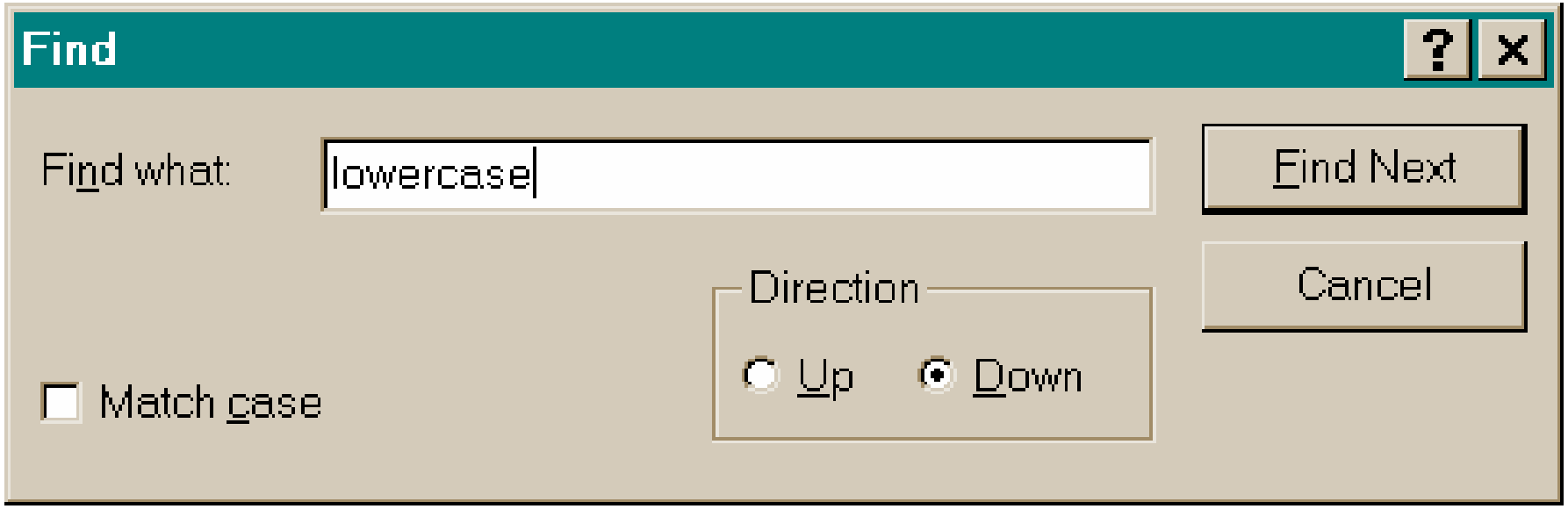
Now consider testing every possible pair, but testing many pairs simultaneously.

We are creating test cases that combine all variables at once, and that assure that every value of every variable is paired with every other value of every other variable.

Each of these test cases covers 21 pairs. **In general, each test case that combines N variables includes N (N-1) / 2 pairs**

	Toolbars PTB	On Startup, Browser Y/N	On Startup, Mail Y/N	On Startup, News Y/N	Start With BVE	Links DU	Followed NE
Test #1	P	Y	Y	Y	B	D	N
2	P	Y	N	N	V	D	E
3	P	N	Y	Y	E	U	E
4	T	Y	N	Y	E	U	N
5	T	N	Y	N	B	D	E
6	T	N	N	Y	V	D	N
7	B	Y	N	N	B	U	E
8	B	N	Y	Y	V	U	E
9	B	N	N	N	E	D	N

Combinations Exercise



Here is a simple Find dialog. It takes three inputs:

- `Find what:` a text string
- `Match case:` yes or no
- `Direction:` up or down

Simplify this by considering only three values for the text string, "lowercase" and "Mixed Cases" and "CAPITALS".

Combinations Exercise

1 How many combinations of these three variables are possible?

2 List ALL the combinations of these three variables.

3 Now create combination tests that cover all possible pairs of values, but don't try to cover all possible triplets. List one such set.

4 How many test cases are in this set?

Combinations Exercise

Find has three cases:

L M C (lower, mixed, capitals)

Match has two cases:

Y N (yes, no)

Direction has two cases:

U D (up, down)

1. **Total cases is 3 x 2 x 2 = 12**

2. **Full set has 12 tests**

L Y U M Y U C Y U

L Y D M Y D C Y D

L N U M N U C N U

L N D M N D C N D

3. **A reduced set is**

L Y U

L N D

M Y D

M N U

C Y U

C N D

4. **The total is 6**

Combination Testing

Imagine a program with 3 variables, V1 has 3 possible values, V2 has 2 possible values and V3 has 2 possible values.

If V1 and V2 and V3 are independent, the number of possible combinations is 12 (3 x 2 x 2)

Building a simple combination table:

- Label the columns with the variable names, listing variables in descending order (of number of possible values)
- Each column (before the last) will have repetition. Suppose that A, B, and C are in column K of N columns. To determine how many times (rows in which) to repeat A before creating a row for B, multiply the number of variable values in columns K+1, K+2, . . ., N.

Combination Testing

Building an all-pairs combination table:

- Label the columns with the variable names, listing variables in descending order (of number of possible values)
- If the variable in column 1 has V1 possible values and the variable in column 2 has V2 possible values, then there will be at least V1 x V2 rows (draw the table this way but leave a blank row or two between repetition groups in column 1).
- 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, C and V2 is 2, then column 1 would contain A, A, blank row, B, B, blank row, C, C, blank row.

Combination Testing

Building an all-pairs combination table:

- In the second column, list all the values of the variable, skip the line, list the values, etc. For example, if variable 2's possible values are X,Y, then the table looks like this so far

A	X
A	Y
B	X
B	Y
C	X
C	Y

Combination Testing

Building an all-pairs combination table:

- Each section of the third column (think of AA as defining a section, BB as defining another, etc.) will have to contain every value of variable 3. Order the values such that the variables also make all pairs with variable 2.
- Suppose variable 2 can be 1,0
- The third section can be filled in either way, and you might highlight it so that you can reverse it later. The decision (say 1,0) is arbitrary.

A	X	1
A	Y	0
B	X	0
B	Y	1
C	X	
C	Y	

Now that we've solved the 3-column exercise, let's try adding more variables. Each of them will have two values.

Combination Testing

The 4th column went in easily (note that we started by making sure we hit all pairs of values of column 4 and column 2, then all pairs of column 4 and column 3.

Watch this first attempt on column 5. We achieve all pairs of GH with columns 1, 2, and 3, but miss it for column 4.

The most recent arbitrary choice was HG in the 2nd section. (Once that was determined, we picked HG for the third in order to pair H with a 1 in the third column.)

So we will erase the last choice and try again:

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

Combination Testing

We flipped the last arbitrary choice (column 5, section 2, to GH from HG) and erased section 3. We then fill in section 3 by checking for missing pairs. GH, GH gives us two XG, XG pairs, so we flip to HP for the third section and have a column 2 X with a column 5 H and a column 2 Y with a column 5 G as needed to obtain all pairs.

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

Combination Testing

But when we add the next column, we see that we just can't achieve all pairs with 6 values. The first one works up to column 4 but then fails to get pair EJ or FI. The next fails on GJ, HI

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

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

Combination Testing

When all else fails, add rows. We need one for GJ and one for HI, so add two rows. In general, we would need as many rows as the last column has values.

The other values in the two rows are arbitrary, leave them blank and fill them in as needed when you add new columns. At the very end, fill the remaining blank ones with arbitrary values

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

Combination Testing

If a variable is continuous but maps to a number line, partition and use boundaries as the distinct values under test. If all variables are continuous, we end up with all pairs of all boundary tests of all variables. We don't achieve all triples, all quadruples, etc.

If some combinations are of independent interest, add them to the list of n-tuples to test.

- With the six columns of the example, we reduced 96 tests to 8. Give a few back (make it 12 or 15 tests) and you still get enormous reduction.
- Examples of "independent interest" are known (from tech support) high risk cases, cases that jointly stress memory, configuration combinations (Var 1 is operating systems, Var 2 is printers, etc.) that are prevalent in the market, etc.

Combination: Interesting Reading

- Cohen, Dalal, Parelius, Patton,"The Combinatorial Design Approach to Automatic Test Generation",IEEE Software, Sept. 96 http://www.argreenhouse.com/papers/gcp/AETGissre96.shtml
- Cohen, Dalal, Fredman, Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design", IEEE Trans on SW Eng. Vol 23#7, July 97 http://www.argreenhouse.com/papers/gcp/AETGieee97.shtml
- **Several other papers on AETG are available at** http://aetgweb.argreenhouse.com/papers.html
- Also interesting: a discussion of orthogonal arrays http://www.stsc.hill.af.mil/CrossTalk/1997/oct/planning.html

Stochastic or Random Testing

By this point, we have undoubtedly run out of time in the quality week tutorial. For more information on this topic, see my paper, Architectures of Test Automation, http://www.kaner.com/testarch.html