

Black Box Software Testing

Pacific Northwest Software Quality Conference 2003

by

Cem Kaner, J.D., Ph.D.
Professor of Software Engineering
Florida Institute of Technology
and

James Bach
Principal, Satisfice Inc.

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

Black Box Test Design

THE WORKSHOP ANNOUNCEMENT

- This workshop surveys several popular approaches to black box software testing, looks at tradeoffs involved in designing good tests, then focuses on a few approaches: primarily exploratory, domain, scenario, and regression testing. We also work through the basics of combination testing, and look at the role (and types) of oracles in the design of automated tests.
- Please bring a laptop if you have one, and install Open Office (www.openoffice.org) on it. We'll do some exercises that you might find a bit more concrete if you have a copy of Open Office in front of you.

ACKNOWLEDGEMENTS AND INTELLECTUAL PROPERTY NOTES

- These notes were originally developed in co-authorship with Hung Quoc Nguyen. James Bach has contributed substantial material. We also thank Jack Falk, Elizabeth Hendrickson, Doug Hoffman, Bob Johnson, Brian Lawrence, Melora Svoboda, Ross Collard, Pat McGee, and the participants in the Los Altos Workshops on Software Testing and the Software Test Managers' Roundtables. Additional acknowledgements appear on specific slides.
- These course notes are copyrighted. You may not make additional copies of these notes without permission. For educational use, you can obtain a license from Kaner's lab's website, www.testingeducation.org. For commercial copying, request permission from Cem Kaner, kaner@kaner.com.
- The practices recommended and discussed in this course are useful for an introduction to testing, but more experienced testers will adopt additional practices. We are writing this course with basic commercial software development in mind. Mission-critical and life-critical software development efforts involve specific and rigorous procedures that are not described in this course.

About Cem Kaner

My current job titles are Professor of Software Engineering at the Florida Institute of Technology, and Research Fellow at Satisfice, Inc. I'm also an attorney, whose work focuses on same theme as the rest of my career: satisfaction and safety of software customers and workers.

I've worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality. These have provided many insights into relationships between computers, software, developers, and customers.

I'm the senior author of three books:

- *Lessons Learned in Software Testing* (with James & Bret Pettichord)
- *Bad Software* (with David Pels)
- *Testing Computer Software* (with Jack Falk & Hung Quoc Nguyen).

I studied Experimental Psychology for my Ph.D., with a dissertation on Psychophysics (essentially perceptual measurement). This field nurtured my interest in *human factors* (and thus the usability of computer systems) and in *measurement theory* (and thus, the development of valid software metrics.)



About James Bach



I started in this business as a programmer. I like programming. But I find the problems of software quality analysis and improvement more interesting than those of software production. For me, there's something very compelling about the question "How do I know my work is good?" Indeed, how do I know anything is good? What does good mean? That's why I got into SQA, in 1987.

Today, I work with project teams and individual engineers to help them plan SQA, change control, and testing processes that allow them to understand and control the risks of product failure. I also assist in product risk analysis, test design, and in the design and implementation of computer-supported testing. Most of my experience is with market-driven Silicon Valley software companies like Apple Computer and Borland, so the techniques I've gathered and developed are designed for use under conditions of compressed schedules, high rates of change, component-based technology, and poor specification.

Black Box Software Testing

Part 1

ORACLES

Does font size work in Open Office?



What's your oracle?

Oracles

An oracle is the principle or mechanism by which you recognize a problem.

“..it works”

really means...

“...it appeared to meet some requirement to some degree.”

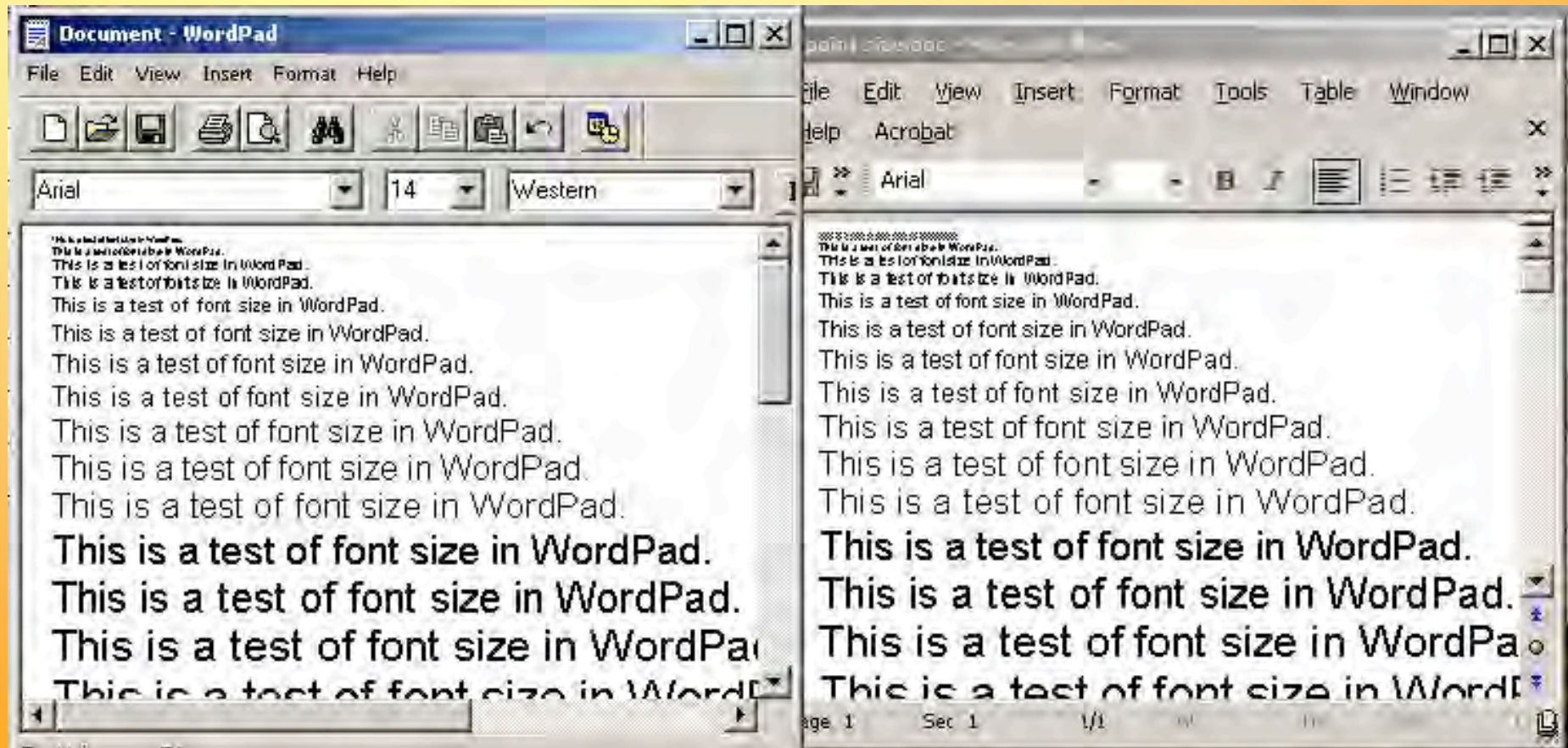
OK, so what about WordPad?



What if we compared WordPad to Word?

WordPad

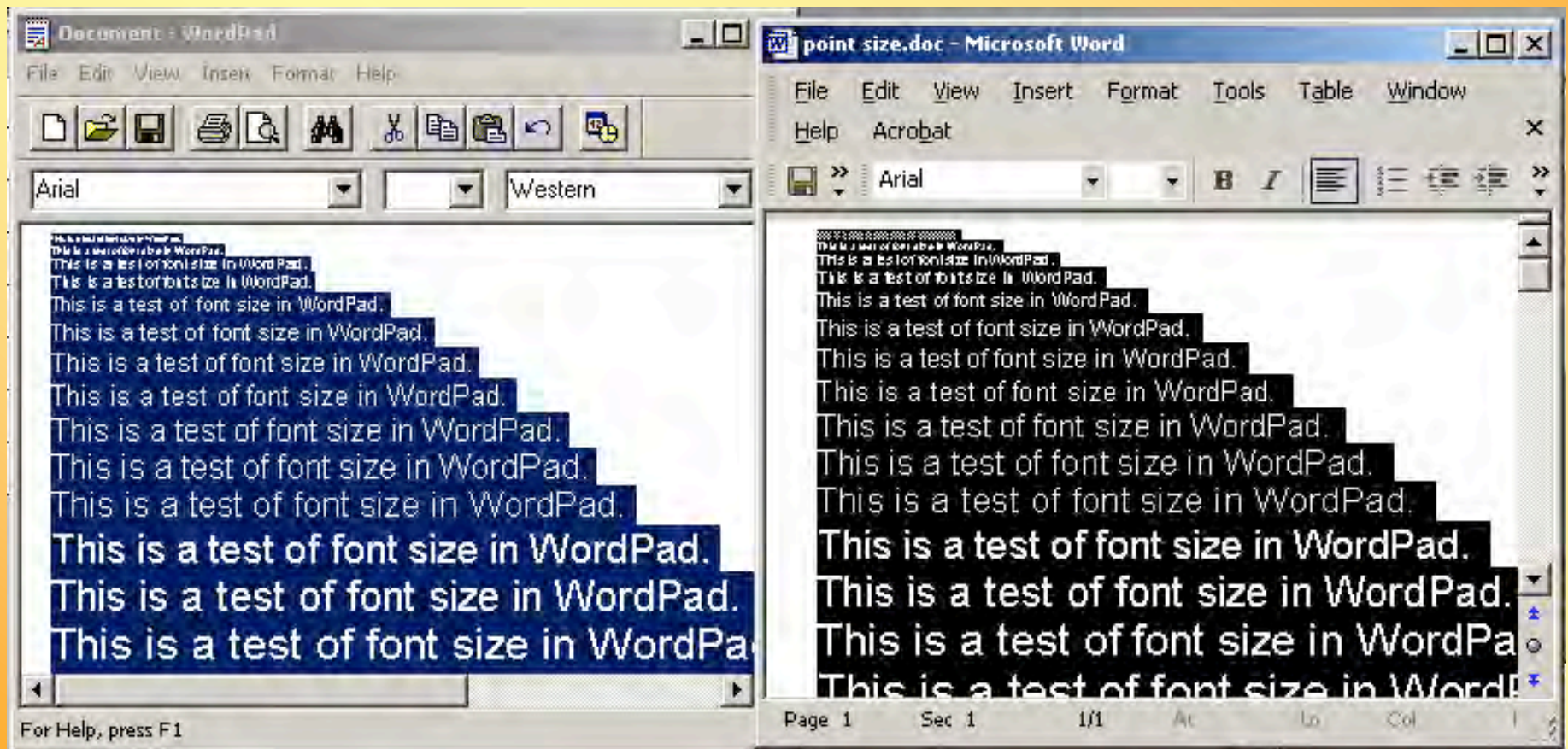
Word



What does this tell us?

WordPad

Word



Some useful oracle heuristics

- **Consistent with History:** Present function behavior is consistent with past behavior.
- **Consistent with our 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 documented or advertised behavior.
- **Consistent with Specifications or Regulations:** Function behavior is consistent with claims that must be met.
- **Consistent with User's Expectations:** 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 apparent purpose.

Testing is about ideas. Heuristics give you ideas.

- A heuristic is a fallible idea or method that may help solve a problem.
- You don't comply with a heuristic; you apply it. Heuristics can hurt you when elevated to the status of authoritative rules.
- Heuristics represent wise behavior only in context. They do not contain wisdom.
- Your relationship to a heuristic is the key to applying it wisely.

“Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem.”

- George Polya, *How to Solve It*

Some font size testing issues...

- What to cover?
 - Every font size (to a tenth of a point).
 - Every character in every font.
 - Every method of changing font size.
 - Every user interface element associated with font size functions.
 - Interactions between font sizes and other contents of a document.
 - Interactions between font sizes and every other feature in Wordpad.
 - Interactions between font sizes and graphics cards & modes.
 - Print vs. screen display.
- What's your oracle?
 - What do you know about typography?
 - Definition of “point” varies. There are as many as six different definitions (<http://www.oberonplace.com/dtp/fonts/point.htm>)
 - Absolute size of characters can be measured, but not easily (<http://www.oberonplace.com/dtp/fonts/fontsize.htm>)
 - How closely must size match to whatever standard is chosen?
 - Heuristic approaches: relative size of characters; comparison to MS Word.
 - Expectations of different kinds of users for different uses.

Risk as a simplifying factor

- For **Wordpad**, we don't care if font size meets precise standards of typography!
- In general it can vastly simplify testing if we focus on whether the product has a problem that matters, rather than whether the product merely satisfies all relevant standards.
- Effective testing requires that we understand standards as they relate to how our clients value the product.

Instead of thinking **pass** vs. **fail**,
Consider thinking **problem** vs. **no problem**.

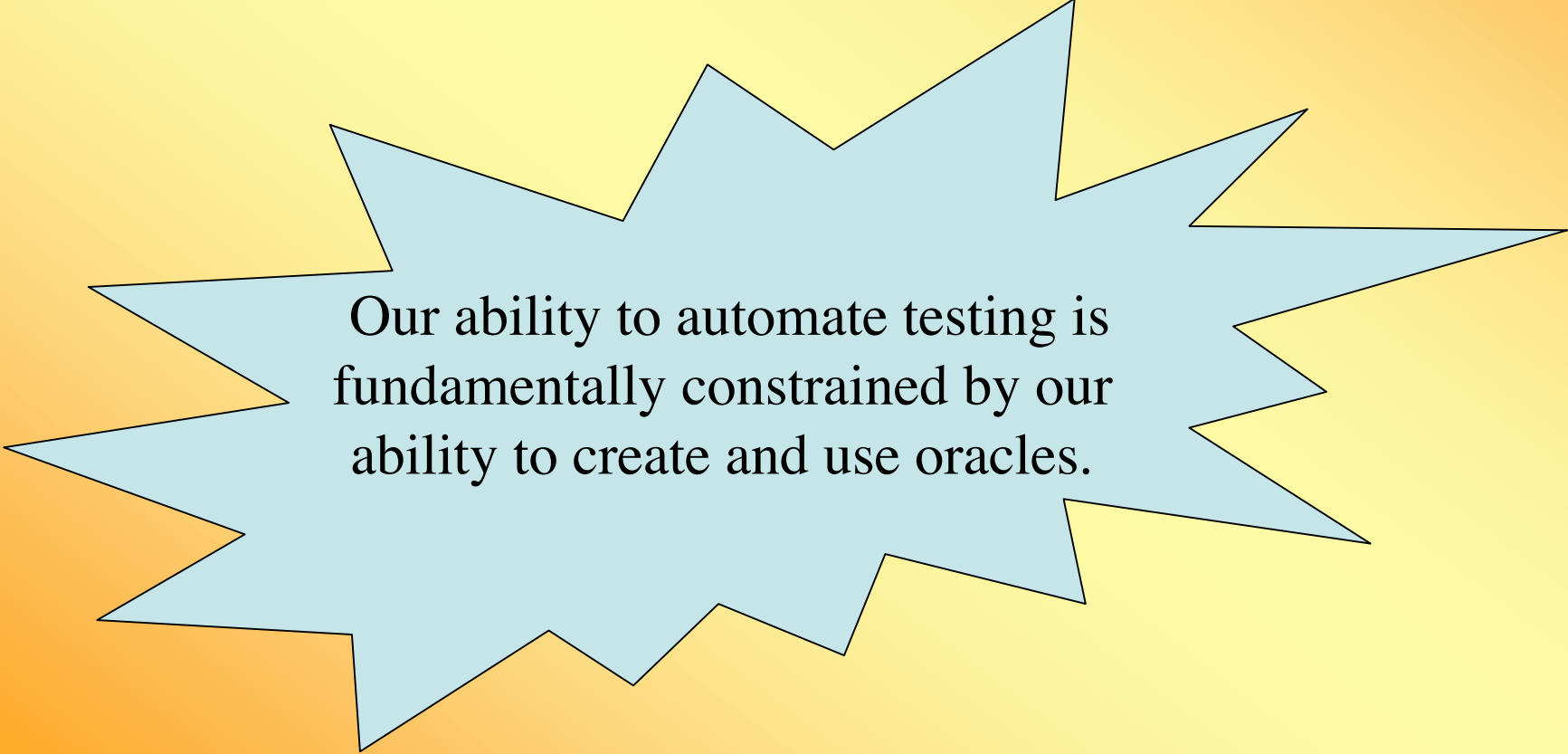
Risk as a simplifying factor

- What if we applied the same evaluation approach
 - that we applied to WordPad
 - to Open Office or MS Word or Adobe PageMaker?

The same evaluation criteria lead to
different conclusions in different contexts

The oracle problem and test automation

- We often hear that all testing should be automated.
- Automated testing depends on our ability to programmatically detect when the software under test fails a test.

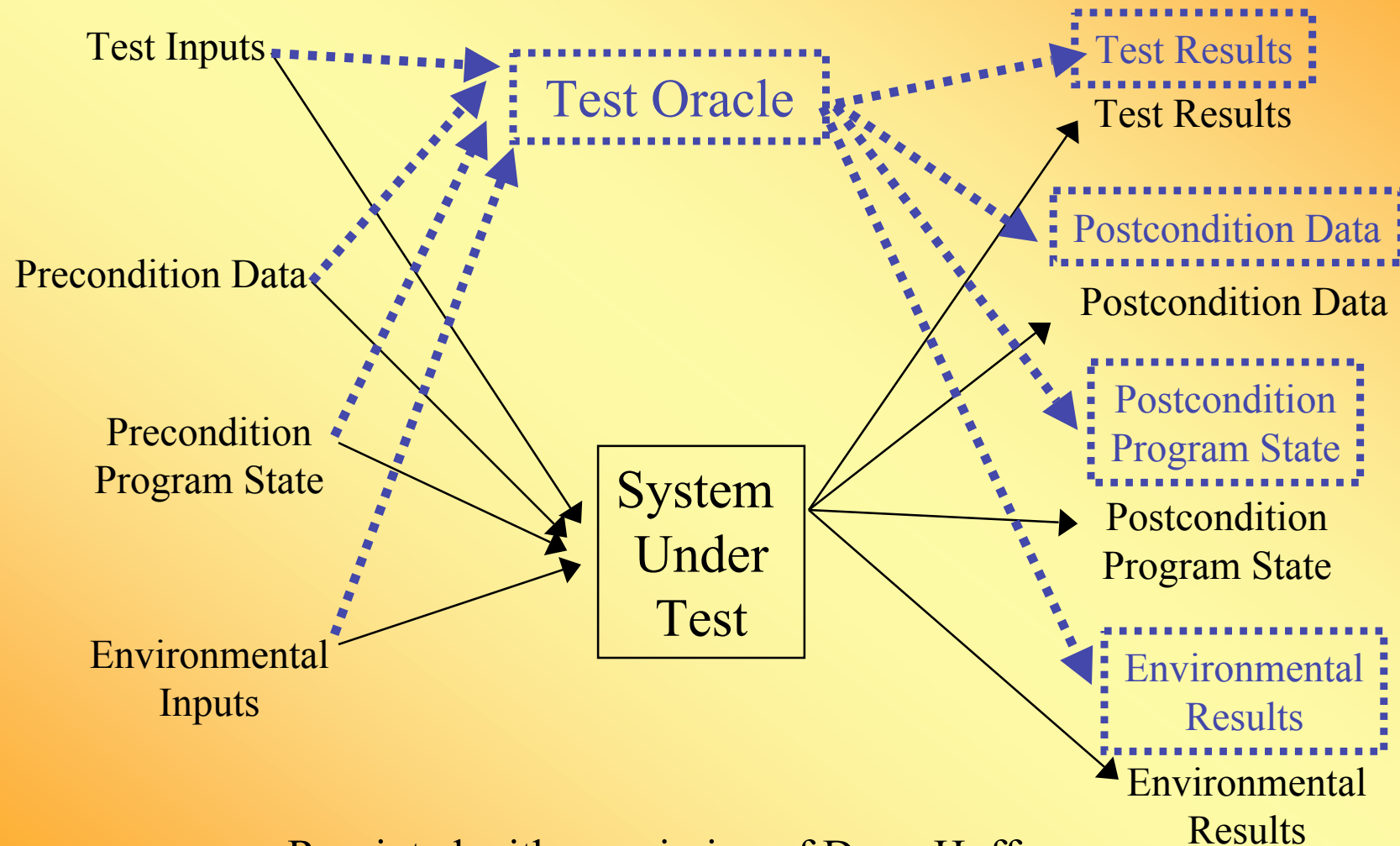


Our ability to automate testing is fundamentally constrained by our ability to create and use oracles.

The oracle problem and automation

- One common way to define an oracle is as a source of expected results.
 - Under this view, you compare your results to those obtained from the oracle. If they match, the program passes. If they don't match, the program fails.
 - The comparison between MS WordPad and MS Word illustrates this approach.
 - ***It is important to recognize that this evaluation is heuristic:***
 - **We can have false alarms:** A mismatch between WordPad and Word might not matter.
 - **We can miss bugs:** A match between WordPad and Word might result from the same error in both programs.

Oracle comparisons are heuristic: We compare only a few result attributes



Reprinted with permission of Doug Hoffman

Automated tests narrow the oracle's scope

- An automated test is not equivalent to the most similar manual test:
 - The mechanical comparison is typically more precise (and will be tripped by irrelevant discrepancies)
 - The skilled human comparison will sample a wider range of dimensions, noting oddities that one wouldn't program the computer to detect.

Black Box Software Testing

Part 2

DOMAIN TESTING

Domain testing

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

Let's work a simple example

Here is a program's specification:

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

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

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

Working through the example

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

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

Refer to Testing Computer Software, Chapter 1

1. The simple, mainstream tests

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

Here is the screen display that results from that test.

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



? 3
? 7
10
? _

Refer to Testing Computer Software, Chapter 1

2. Test each function sympathetically

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

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

3. Test broadly before deeply

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

4. Look for more powerful tests

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

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

4. Look for more powerful tests

What?

Why?

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

5. Reducing the testing burden

There are $199 \times 199 = 39,601$ test cases for valid values:

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

199 values per variable

$199 \times 199 = 39,601$ possible tests

So should we test them all?

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

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

Why?

5. Equivalence class & boundary analysis

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

5. Equivalence class & boundary analysis

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

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

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

5. Classical equivalence class and boundary analysis

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

– Myers, Art of Software Testing, 45

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

5. Myers' boundary table

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

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

5. Myers' boundary table

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

But we routinely consider additional cases, such as special cases that are inside the range (e.g. 0) and errors on a different dimension from your basic "too big" and "too small".

5. Myers' boundary table

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

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

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

Boundary table as a test plan component

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

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

Building the table (in practice)

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

Black Box Software Testing

DOMAIN TESTING and Reusable Test Matrices

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.

Using Test Matrices for Routine Issues

- A matrix is a concise organizer of simple tests, especially useful for function tests and domain tests
- The matrix groups 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.
- Matrix structure:
 - 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.

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.)

Typical Uses of Test Matrices

- You could create a matrix for almost any type of variable. For example, imagine listing all of the hardware (including connection, power, etc.) error conditions that could cause failure of a file save operation.
- You can often re-use a matrix 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.

Error Handling when Writing a File

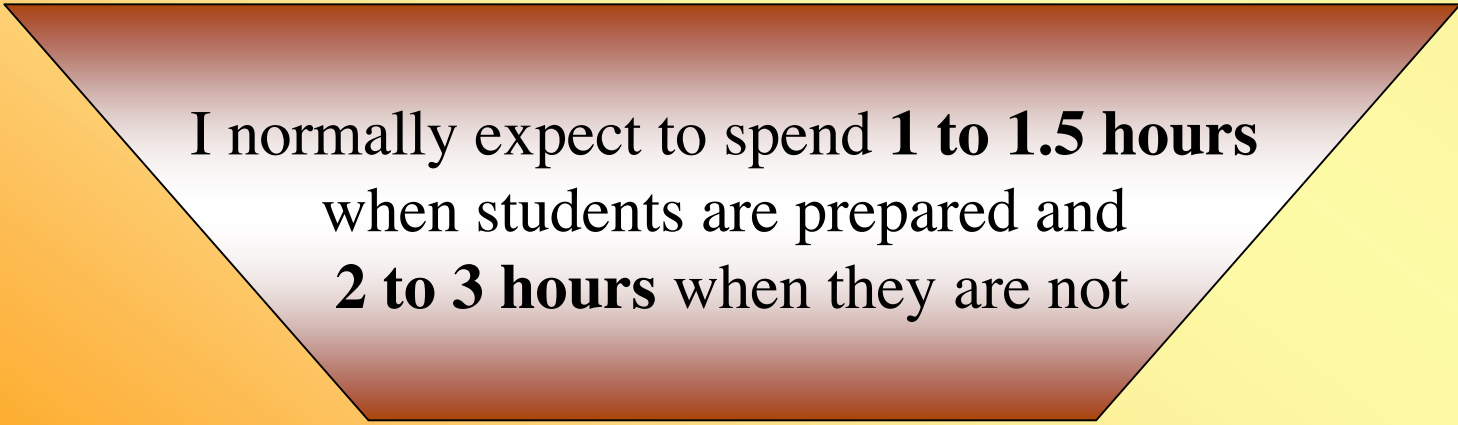
- full local disk
- almost full local disk
- write protected local disk
- damaged (I/O error) local disk
- unformatted local disk
- remove local disk from drive after opening file
- timeout waiting for local disk to come back online
- keyboard and mouse I/O during save to local disk
- other interrupt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

Homework: File Name Matrix

- Tomorrow, we'll illustrate the process of creating test matrices by brainstorming a specific one.

(To be determined on a class-by-class basis)

- Please spend 15 minutes at home writing a list of file name tests. Bring your notes with you, write your name on them and hand a copy of them in at the start of class.
- -----
- We'll take up the assignment with a brainstorming session.



I normally expect to spend **1 to 1.5 hours**
when students are prepared and
2 to 3 hours when they are not

Matrix Construction Brainstorm

Brainstormers' Rules

- Don't criticize others' contributions
- Jokes are OK, and are often valuable
- Goal is to get lots of ideas, filter later.
- Recorder and facilitator keep their opinions to themselves.

Matrix Construction Brainstorm

Facilitators' and Recorders' Rules

- Exercise patience: Goal is to get lots of ideas.
- Encourage non-speakers to speak.
- Use multiple colors when recording
- Echo the speaker's words.
- Record the speaker's words
- The rule of three 10's. Silence is OK.
- Switch levels of analysis.

Some references:

- S. Kaner, Lind, Toldi, Fisk & Berger, Facilitator's Guide to Participatory Decision-Making
- Freedman & Weinberg, Handbook of Walkthroughs, Inspections & Technical Reviews
- Doyle & Straus, How to Make Meetings Work.

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.

Black Box Software Testing

DOMAIN TESTING

Understanding Domain Testing

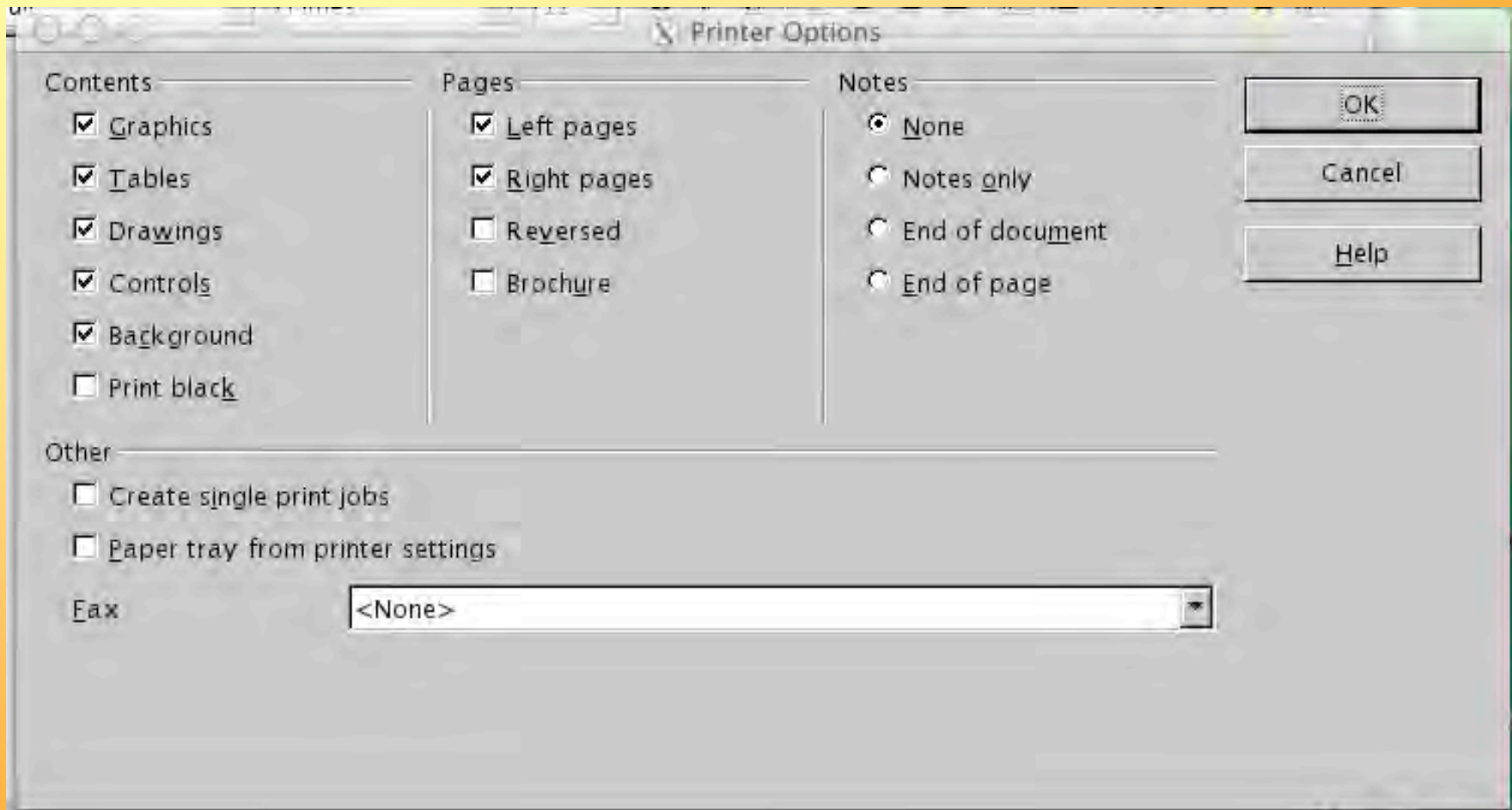
Simple Exercises



This is the print dialog in Open Office. Suppose that

1. The largest number of copies you could enter in Number of Copies field is 999, OR
 2. Your printer will manage multiple copies, for up to 99 copies.
- For each case, do a traditional domain analysis

Domain analysis on these variables?



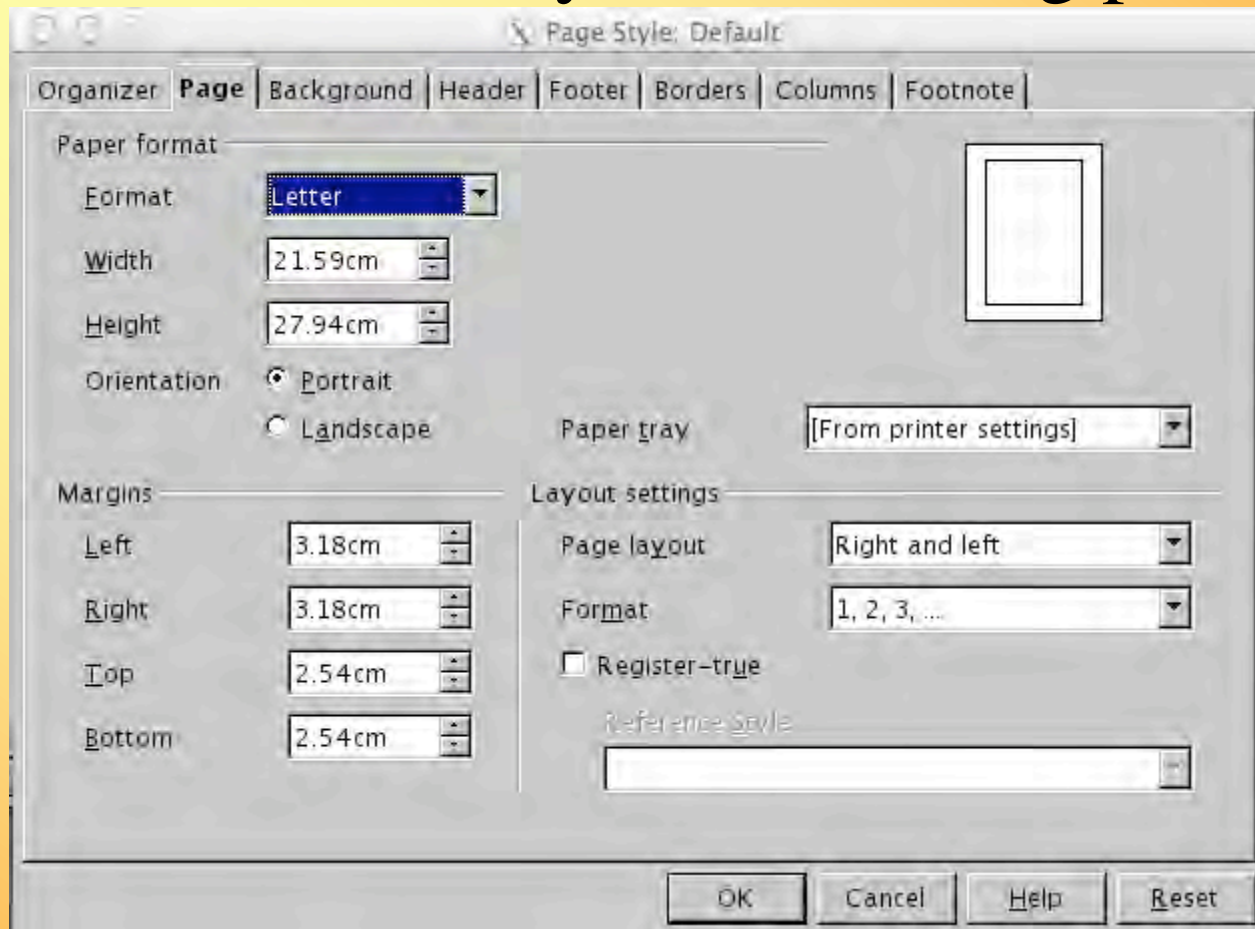
The image shows a 'Printer Options' dialog box with the following sections and variables:

- Contents:**
 - ☒ Graphics
 - ☒ Tables
 - ☒ Drawings
 - ☒ Controls
 - ☒ Background
 - ☐ Print black
- Pages:**
 - ☒ Left pages
 - ☒ Right pages
 - ☐ Reversed
 - ☐ Brochure
- Notes:**
 - ☒ None
 - ☐ Notes only
 - ☐ End of document
 - ☐ End of page
- Other:**
 - ☐ Create single print jobs
 - ☐ Paper tray from printer settings
- Fax:** A dropdown menu currently showing '<None>'.

Buttons on the right: OK, Cancel, Help.

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

Domain analysis on floating point



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

Exercise

For each of the following,

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

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

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

Understanding domain testing

- People were treating values as equivalent long before anyone proposed a theoretical description of domain testing.
- The most important idea in domain testing is that it provides a sensible basis for sampling from a domain.
- Definition: Domain
 - In mathematics,
 - The domain of a function is the set of all input values over which the function is defined.
 - The range (or output domain) of the function is the set of all values that the function can produce.
 - Early descriptions of domain testing focused on inputs, but we routinely applied the analysis to outputs (Testing Computer Software, 1st edition, 1988, reflected that practice.)

Understanding domain testing

*In domain testing, we
partition a domain
into sub-domains (equivalence classes)
and then test
using values from each sub-domain.*

Understanding domain testing

1. What is equivalence?

4 views of what makes values equivalent. Each has practical implications

- ***Intuitive Similarity***: two test values are equivalent if they are so similar to each other that it seems pointless to test both.
 - This is the earliest view and the easiest to teach
 - Little guidance for subtle cases or multiple variables
- ***Specified As Equivalent***: two test values are equivalent if the specification says that the program handles them in the same way.
 - Testers complain about missing specifications may spend enormous time writing specifications
 - Focus is on things that were specified, but there might be more bugs in the features that were un(der)specified

Understanding domain testing

1. What is equivalence?

- ***Equivalent Paths:*** two test values are equivalent if they would drive the program down the same path (e.g. execute the same branch of an IF)
 - Tester should be a programmer
 - Tester should design tests from the code
 - Some authors claim that a complete domain test will yield a complete branch coverage.
 - No basis for picking one member of the class over another.
 - Two values might take program down same path but have very different subsequent effects (e.g. timeout or not timeout a subsequent program; or e.g. word processor's interpretation and output may be the same but may yield different interpretations / results from different printers.)
- ***Risk-Based:*** two test values are equivalent if, given your theory of possible error, you expect the same result from each.
 - Subjective analysis, differs from person to person. It depends on what you expect (and thus, what you can anticipate).
 - Two values may be equivalent relative to one potential error but non-equivalent relative to another.

Understanding domain testing

2. Test which values from the equivalence class?

Most discussions of domain testing start from several assumptions:

- (a) The domain is continuous [This is easily relaxed -- CK]
- (b) The domain is linearizable (members of the domain can be mapped to the number line) or, at least, the domain is an ordered set (given two elements, one is larger than the other or they are equal)
- (c) The comparisons that cause the program to branch are simple, linear inequalities

"It is possible to move away from these assumptions, but the cost can be high." --- Clarke, Hassell, & Richardson, p. 388

- If we think in terms of paths, can we use any value that drives the program down the correct path? This approach is common in coverage-focused testing. Unfortunately, it doesn't yield many failures. See Hamlet & Taylor
- If you can map the input space to a number line, then boundaries mark the point or zone of transition from one equivalence class to another. These are said to be good members of equivalence classes to use ***because the program is more likely to fail at a boundary.***

Understanding domain testing

2. Test which values from the equivalence class?

The program is more likely to fail at a boundary?

- *Suppose program design:*
 - INPUT < 10 result: Error message
 - 10 <= INPUT < 25 result: Print "hello"
 - 25 <= INPUT result: Error message
- *Some error types*
 - Program doesn't like numbers
 - Any number will do
 - Inequalities mis-specified (e.g. INPUT <= 25 instead of < 25)
 - Detect only at boundary
 - Boundary value mistyped (e.g. INPUT < 52, transposition error)
 - Detect at boundary and any other value that will be handled incorrectly
- Boundary values (here, test at 25) catch all three errors
- Non-boundary values (consider 53) may catch only one of the three errors

Understanding domain testing

2. Test which values from the equivalence class?

- The emphasis on boundaries is inherently risk-based
- But the explicitly risk-based approach goes further
 - Consider many different risks
 - Partitioning driven by risk
 - Selection of values driven by risk:
 - A member of an equivalence class is a **best representative** (relative to a potential error) if no other member of the class is more likely to expose that error than the best representative.
 - » Boundary values are often best representatives
 - » We can have best representatives that are not boundary values
 - » We can have best representatives in non-ordered domains

Risk-based equivalence

- Consider these cases. Are these paired tests equivalent?

– If you tested Would you test

51+52

52+53

53+54

54+55

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

Risk-based equivalence

- Given the following potential error:

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

Another example: Non-obvious boundaries

	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

Another example of non-obvious boundaries

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

In sum: equivalence classes and representative values

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.

In an ordered set, 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.

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. We call those most powerful tests the best representatives of the class.

Xref: stratified sampling: http://www.wikipedia.org/wiki/Stratified_sampling

A new boundary and equivalence table

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

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

Examples of ordered sets

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

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

Examples of ordered sets

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

Examples of ordered sets

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

Non-ordered sets

- ***A sample problem:***
 - *There are about 2000 Windows-compatible printers, plus multiple drivers for each. We can't test them all.*
- ***They are not ordered, but maybe we can form equivalence classes and choose best representatives anyway.***
- Here are two examples from programs (desktop publishing and an address book) developed in 1991-92.

Non-ordered sets

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?

Non-ordered sets

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.

We pick different best representatives from the same equivalence class, depending on which error we are trying to detect.

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.)

More examples of non-ordered sets

- Here are more examples of variables that don't fit the traditional mold for equivalence classes but which have enough values that we will have to sample from them. What are the boundary cases here?
- Membership in a common group
 - Such as employees vs. non-employees.
 - Such as workers who are full-time or part-time or contract.
- Equivalent hardware
 - such as compatible modems, video cards, routers
- 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

Interactions among variables

Rather than thinking about a single variable with a single range of values, a variable might have different ranges, such as the day of the month, in a date:

1-28

1-29

1-30

1-31

We analyze the range of dates by partitioning the month field for the date into different sets:

{February}

{April, June, September, November}

{Jan, March, May, July, August, October, December}

For testing, you want to pick one of each. There might or might not be a “boundary” on months. The boundaries on the days, are sometimes 1-28, sometimes 1-29, etc

» This is nicely analyzed by Jorgensen:
» Software Testing--A Craftsman's Approach.

Another example of interaction

- Interaction-thinking is important when we think of an output variable whose value is based on some input variables. Here's an example that gives students headaches on tests:
- 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

Domain Testing

- Strengths
 - Find highest probability errors with a relatively small set of tests.
 - Intuitively clear approach, easy to teach and understand
 - Extends well to multi-variable situations
- Blind spots or weaknesses
 - Errors that are not at boundaries or in obvious special cases.
 - The "competent programmer hypothesis" can be misleading.
 - Also, the actual domains are often unknowable.
 - Reliance on best representatives for regression testing leads us to overtest these cases and undertest other values that were as, or almost as, good.

Black Box Software Testing

DOMAIN TESTING and 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

In a combination test, we test several variables together. Each test explicitly sets values for each of the variables under test.

Combination Testing

- There are several approaches to combination testing:
 - ***Mechanical (or procedural)***. The tester uses a routine procedure to determine a good set of tests
 - ***Risk-based***. The tester combines test values (the values of each variable) based on perceived risks associated with noteworthy combinations
 - ***Scenario-based***. The tester combines test values on the basis of interesting stories created for the combinations

Mechanical approach #1

"Weak testing" version 1

- Suppose that we have three numeric variables, V1, V2 and V3.
- We analyze each variable in terms of its subdomains and boundaries. Thus we might have for each variable:
 - V1: Too-low (TL), lowest valid (LV), biggest valid (BV), too big (TB)
 - V2: Too-low (TL), lowest valid (LV), biggest valid (BV), too big (TB)
 - V3: Too-low (TL), lowest valid (LV), biggest valid (BV), too big (TB)
- In the "weak" combination, we create enough tests to include all of the values of each variable, such as:

	V1	V2	V3
Test 1	TL	TL	TL
Test 2	LV	LV	LV
Test 3	BV	BV	BV
Test 4	TB	TB	TB

This is an example of "all singles" testing. (More on that later.)

Mechanical approach #1

"Weak testing" versions 2 and 3

- In the first version, we considered values only on one dimension. Thus, we don't consider error cases on the other dimensions (such as null, non-number, etc.)
- In the second version, we include all of the "interesting" values. Otherwise, the analysis is the same.
- In the third version, we treat error cases specially:
 - We generate a core set of tests for "valid" (non-error) inputs
 - We generate additional tests in which one error case is allowed per test case

Mechanical approach #2

"Strong testing" version 1

- Consider again V1, V2, and V3, with values of interest:
 - Too-low (TL), lowest valid (LV), biggest valid (BV), too big (TB)
- In Jorgensen's version of "strong testing", we test every combination of these values:

	V1	V2	V3
Test 1	TL	TL	TL
Test 2	TL	TL	LV
Test 3	TL	TL	BV
Test 4	TL	TL	TB
Test 5	TL	LV	TL
Test 6	TL	LV	LV
Test 7	TL	LV	BV
Test 8	TL	LV	TB
Test 9	TL	BV	TL
Test 10	TL	BV	LV

This is part of the table. The complete table has $4 * 4 * 4$ tests.

This is an example of "all triples" or "all n-tuples" where "n" is the number of variables we are testing together.

Mechanical approach #2

"Strong testing" version 2

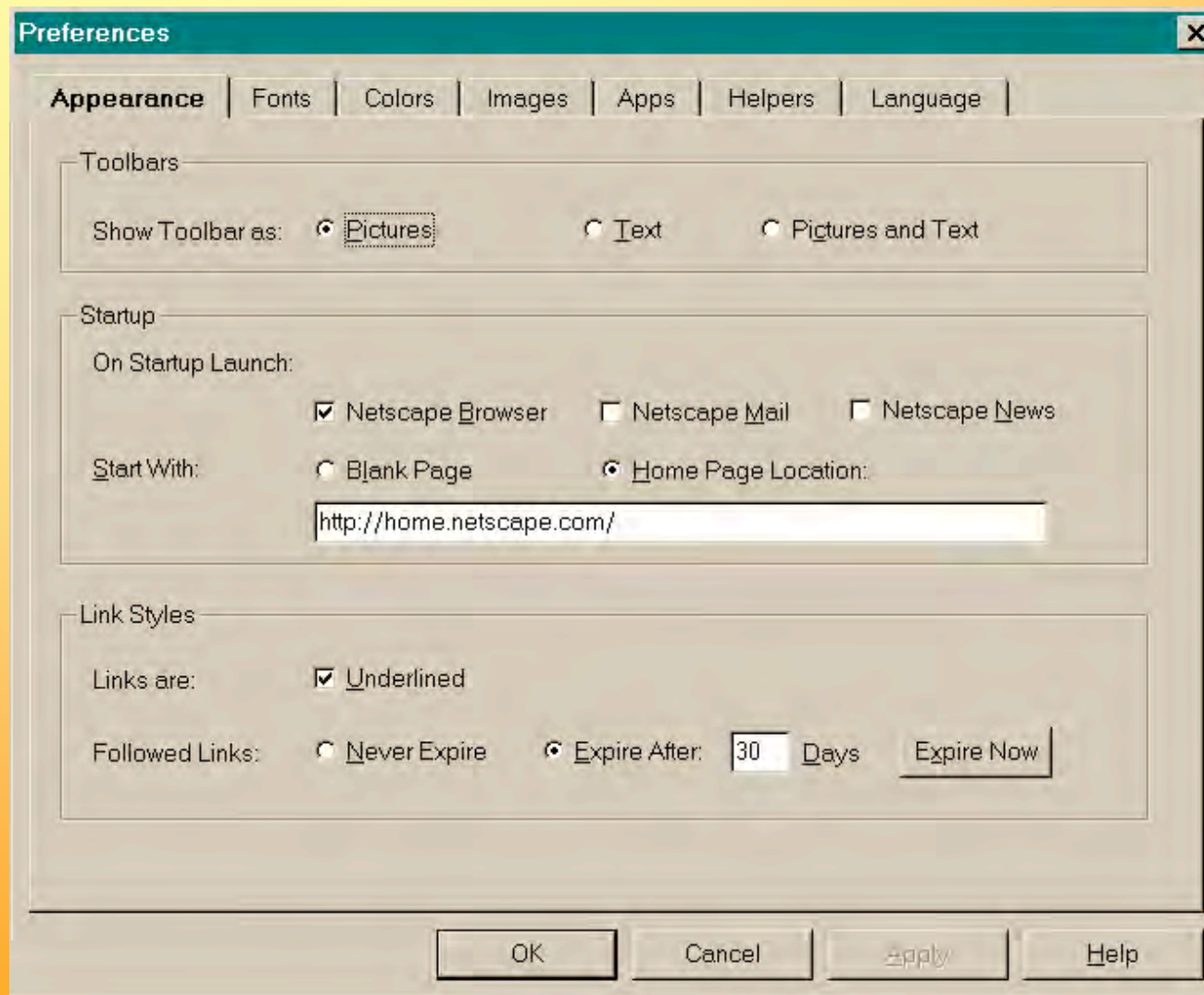
- A common variation restricts combination tests to valid values. Error cases are treated separately in the one-variable tests. In that case, we would exclude TL (too low) and TB (too big), and would have $2 * 2 * 2 = 8$ tests.
- Another variation includes all valid-value combinations plus a separate set of combination tests in which one, some, or all variables have an error value.
- Tests that include several errors are of interest only if we think that multiple errors might have some type of cumulative effect.

Mechanical approach #3

All-pairs testing

- Beizer recommends the all-singles approach (weak testing)
- At one of the LAWST meetings, we were advised that Microsoft often uses a modified all-singles in configuration testing:
 - All singles, plus
 - All other combinations designated by marketing (or by error history) as particularly interesting
- Another approach that requires far fewer tests than all-n-tuples but achieves a systematic manipulation, is all-pairs, which we'll walk through in the next several slides:

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 $3 \times 2 \times 2 \times 2 \times 3 \times 2 \times 2 = 288$ 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	PYYYBDN	PNYYBDN	TTYYPBDN	TNYYBDN	BYYYBDN	BNYYBDN
2	PYYYVDE	PNYYVDE	TTYYPVDE	TNYYVDE	BYYYVDE	BNYYVDE
3	PYYYEDN	PNYYEDN	TTYYPEDN	TNYYEDN	BYYYEDN	BNYYEDN
4	PYYYBUE	PNYYBUE	TTYYPBUE	TNYYBUE	BYYYBUE	BNYYBUE
5	PYYYVUN	PNYYVUN	TTYYPVUN	TNYYVUN	BYYYVUN	BNYYVUN
6	PYYYEUE	PNYYEUE	TTYYPEUE	TNYYEUE	BYYYEUE	BNYYEUE
7	PYYNBDN	PNNYBDN	TYNNBDN	TNNYBDN	BYNNBDN	BNNYBDN
8	PYYNVDE	PNNNVDE	TYNNVDE	TNNNVDE	BYNNVDE	BNNNVDE
9	PYYNEDN	PNNNEDN	TYNNEDN	TNNNEDN	BYNNEDN	BNNNEDN
10	PYYNBUE	PNNNBUE	TYNNBUE	TNNNBUE	BYNNBUE	BNNNBUE
11	PYYNVUN	PNNNVUN	TYNNVUN	TNNNVUN	BYNNVUN	BNNNVUN
12	PYYNEUE	PNNNEUE	TYNNEUE	TNNNEUE	BYNNEUE	BNNNEUE
13	PYYBDE	PNNYBDE	TYNYBDE	TNNYBDE	BYNYBDE	BNNYBDE
14	PYYVDN	PNNYVDN	TYNYVDN	TNNYVDN	BYNYVDN	BNNYVDN
15	PYYEDE	PNNYEDE	TYNYEDE	TNNYEDE	BYNYEDE	BNNYEDE
16	PYYBUN	PNNYBUN	TYNYBUN	TNNYBUN	BYNYBUN	BNNYBUN
17	PYYVUE	PNNYVUE	TYNYVUE	TNNYVUE	BYNYVUE	BNNYVUE
18	PYYEUN	PNNYEUN	TYNYEUN	TNNYEUN	BYNYEUN	BNNYEUN
19	PYYNBDE	PNNNBDE	TYNNBDE	TNNNBDE	BYNNBDE	BNNNBDE
20	PYYNVDN	PNNNVDN	TYNNVDN	TNNNVDN	BYNNVDN	BNNNVDN
21	PYYNEDE	PNNNEDE	TYNNEDE	TNNNEDE	BYNNEDE	BNNNEDE
22	PYYNBUN	PNNNBUN	TYNNBUN	TNNNBUN	BYNNBUN	BNNNBUN
23	PYYNVUE	PNNNVUE	TYNNVUE	TNNNVUE	BYNNVUE	BNNNVUE
24	PYYNEUN	PNNNEUN	TYNNEUN	TNNNEUN	BYNNEUN	BNNNEUN
25	PYNYBDE	PNNYBDE	TYNYBDE	TNNYBDE	BYNYBDE	BNNYBDE
26	PYNYVDN	PNNYVDN	TYNYVDN	TNNYVDN	BYNYVDN	BNNYVDN
27	PYNYEDE	PNNYEDE	TYNYEDE	TNNYEDE	BYNYEDE	BNNYEDE
28	PYNYBUN	PNNYBUN	TYNYBUN	TNNYBUN	BYNYBUN	BNNYBUN
29	PYNYVUE	PNNYVUE	TYNYVUE	TNNYVUE	BYNYVUE	BNNYVUE
30	PYNYEUN	PNNYEUN	TYNYEUN	TNNYEUN	BYNYEUN	BNNYEUN
31	PYNNBDE	PNNNBDE	TYNNBDE	TNNNBDE	BYNNBDE	BNNNBDE
32	PYNNVDN	PNNNVDN	TYNNVDN	TNNNVDN	BYNNVDN	BNNNVDN
33	PYNNEDE	PNNNEDE	TYNNEDE	TNNNEDE	BYNNEDE	BNNNEDE
34	PYNNBUN	PNNNBUN	TYNNBUN	TNNNBUN	BYNNBUN	BNNNBUN
35	PYNNVUE	PNNNVUE	TYNNVUE	TNNNVUE	BYNNVUE	BNNNVUE
36	PYNNNEUN	PNNNNEUN	TYNNNEUN	TNNNNEUN	BYNNNEUN	BNNNNEUN
37	PYNYBDN	PNNYBDN	TYNYBDN	TNNYBDN	BYNYBDN	BNNYBDN
38	PYNYVDE	PNNYVDE	TYNYVDE	TNNYVDE	BYNYVDE	BNNYVDE
39	PYNYEDN	PNNYEDN	TYNYEDN	TNNYEDN	BYNYEDN	BNNYEDN
40	PYNYBUE	PNNYBUE	TYNYBUE	TNNYBUE	BYNYBUE	BNNYBUE
41	PYNYVUN	PNNYVUN	TYNYVUN	TNNYVUN	BYNYVUN	BNNYVUN
42	PYNYEUE	PNNYEUE	TYNYEUE	TNNYEUE	BYNYEUE	BNNYEUE
43	PYNNBDN	PNNNBUN	TYNNBDN	TNNNBUN	BYNNBDN	BNNNBUN
44	PYNNVDE	PNNNVDE	TYNNVDE	TNNNVDE	BYNNVDE	BNNNVDE
45	PYNNEDN	PNNNEDN	TYNNEDN	TNNNEDN	BYNNEDN	BNNNEDN
46	PYNNBUE	PNNNBUE	TYNNBUE	TNNNBUE	BYNNBUE	BNNNBUE
47	PYNNVUN	PNNNVUN	TYNNVUN	TNNNVUN	BYNNVUN	BNNNVUN
48	PYNNNEUE	PNNNNEUE	TYNNNEUE	TNNNNEUE	BYNNNEUE	BNNNNEUE

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

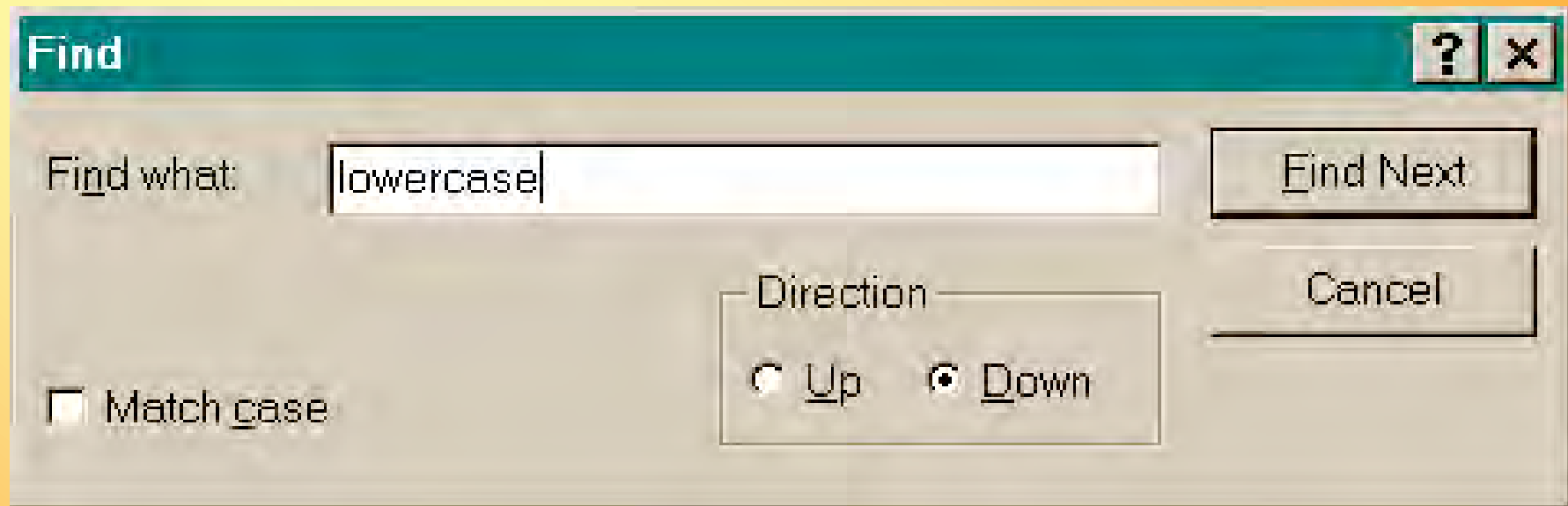
Testing Variables in Combination

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?

The AETG System: An Approach to Testing Based on Combinatorial Design - Netscape

File Edit View Go Communicator Help **PM LIVE**

Bookmarks Netsite <https://aetgweb.tipandring.com/papers/AETGieee97.html>

The AETG System: An Approach to Testing Based on Combinatorial Design

Appeared in July 1997 issue of IEEE Transactions On Software Engineering (Vol. 23, No. 7)

By

D. M. Cohen - IDA-CCS (Work done while at Bellcore.)
S. R. Dalal - Bellcore
M. L. Fredman - Rutgers University
G. C. Patton - Bellcore

TABLE OF CONTENTS

Abstract
1. Introduction
2. The Basic Combinatorial Design Paradigm
3. Logarithmic Growth for n-Way Interaction Testing
4. A Heuristic Algorithm
5. AETG Input Language

5.1 Constraints
5.2 Hierarchy and hierarchical testing

6. Experiments
7. Overview of Applications

7.1 High-Level Test Planning
7.2 Test Case Generation

8. Related Methods
9. Summary
Acknowledgements
References

Abstract

This paper describes a new approach to testing that uses combinatorial designs to generate tests that cover the pair-wise, triple or n-way combinations of a system's test parameters. These are the parameters that determine the system's test scenarios. Examples are system configuration parameters, user inputs and other external events. We implemented this new method in the AETG

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 $3 \times 2 \times 2 = 12$

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 ($3 \times 2 \times 2$)

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 $V1 \times V2$ 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.

Combinatorial 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>
- Also interesting: a discussion of orthogonal arrays
<http://www.stsc.hill.af.mil/CrossTalk/1997/oct/planning.html>
- Beizer, Black Box Testing, Wiley 1995
- Jorgensen, Software Testing: A Craftsman's Approach, 2nd Edition, 2001
- FREE TOOLS:
 - <http://burtleburtle.net/bob/math/jenny.html>
 - <http://www.satisfice.com/tools/pairs.zip>

Combination testing of meaningful relationships

- The combinatorial approach is useful, but with important limitations:
 - It assumes that variables are independent (unrelated)
 - For variables V1 and V2, all values of V2 are possible for any value of V1 (and vice-versa, for V1)
 - For variables V1 and V2, the equivalence classes and best representatives of V2 are the same for any value of V1 (and vice-versa, for V1's classes)
 - Testing is done ONLY with valid values of V1 and V2.
- Cause-effect graphing is one technique (a complex one) for dealing with multiple-variable relationships. If you are interested in this, we suggest studying under Richard Bender.
- The following notes present a less formal approach that we've found useful for guiding exploratory testing of relationships

Combination testing of meaningful relationships

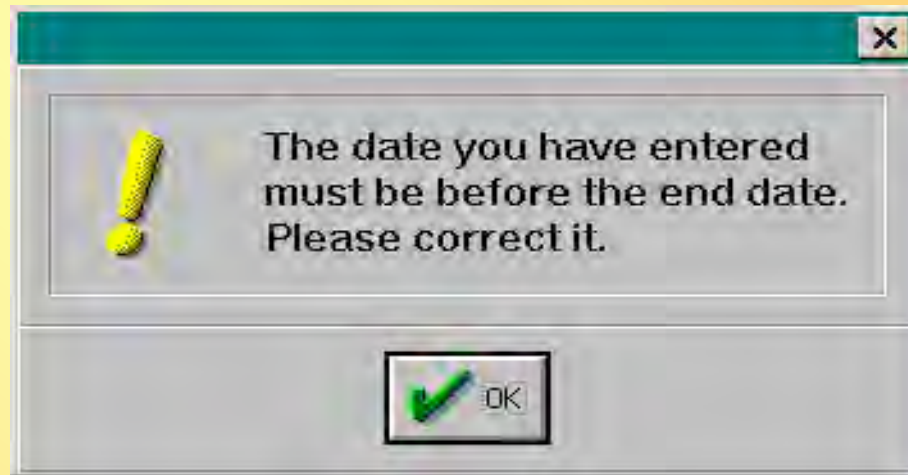
The screenshot shows the TSTimer application window. The menu bar includes File, Edit, Slips, Names, Search, Special, and Help. The main form is titled 'New slip' and has a '+ All' button and a 'Find' field. Below this, there are radio buttons for 'Time Slip' (selected) and 'Expense Slip'. The 'Value' field shows '\$4,475.30'. The form contains several input fields: 'Consultant' (Cem Kaner), 'Client' (softGear Tech), 'Activity' (Research), and 'Description' (Researched the Win 95 documentation). There are also fields for 'Reference', 'Start Date' (5/4/96), 'End Date' (5/3/96), 'Time estimated' (0:00:00), and 'Time spent' (1:40:00). A 'Billing Status' dropdown is set to 'Billable', and there are checkboxes for 'Repeat' and 'Add to Flat Fee'. The 'Rate' field is set to 'Client' with a value of 1 and a rate of 2685.18 (hourly). The 'Override' checkbox is checked. A status bar at the bottom right shows 'Active slips 0'. On the right side of the window, there is a vertical toolbar with buttons for Prev, Next, Go To, New, Save, and Revert.

Field	Value
Consultant	Cem Kaner
Client	softGear Tech
Activity	Research
Description	Researched the Win 95 documentation
Reference	
Start Date	5/4/96
End Date	5/3/96
Time estimated	0:00:00
Time spent	1:40:00
Billing Status	Billable
Repeat	<input type="checkbox"/>
Add to Flat Fee	<input type="checkbox"/>
Rate	Client
Rate Value	1
Rate Rate	2685.18
Rate Unit	(hourly)
Override	<input checked="" type="checkbox"/>

Combination testing of meaningful relationships

- Look at this record, from the Timeslips Deluxe time and billing database. In this dialog box, click the arrow next to the Consultant field to edit the Consultant record (my name, billing info, etc.) or enter a new one.
- If I edit it here, will the changes carry over to every other display of this Consultant record?
- Also, note that the End Date for this task is before the Start Date. *That's not possible.*

Combination testing of meaningful relationships



The program checks the End Date against the Start Date and rejects this pair as impossible because the task can't end before it starts.

*The value of End Date is constrained by Start Date, because End Date *can't be earlier than* Start Date.*

*The value of Start Date constrains End Date, because End Date *can't be earlier than* Start Date.*

Combination testing of meaningful relationships

A relationship table

<i>Field</i>	<i>Entry Source</i>	<i>Display</i>	<i>Print</i>	<i>Related Variable</i>	<i>Relationship</i>
Variable 1 <i>Start date</i>	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 <i>End date</i>	<i>Constraint to a range</i>
Variable 2 <i>End date</i>	Any way you can change values in V2			Variable 1 <i>Start date</i>	<i>Constraint to a range</i>

Tabular Format for Data Relationships

•THE TABLE'S FIELDS

- Field: *Create a row for each field (Consultant, End Date, and Start Date are examples of fields.)*
- Entry Source: *What dialog boxes can you use to enter data into this field? Can you import data into this field? Can data be calculated into this field? List every way to fill the field -- every screen, etc.*
- Display: *List every dialog box, error message window, etc., that can display the value of this field. When you re-enter a value into this field, will the new entry show up in each screen that displays the field? (Not always -- sometimes the program makes local copies of variables and fails to update them.)*
- Print: *List all the reports that print the value of this field (and any other functions that print the value).*
- Related to: *List every variable that is related to this variable. (What if you enter a legal value into this variable, then change the value of a constraining variable to something that is incompatible with this variable's value?)*
- Relationship: *Identify the relationship to the related variable.*

Combination testing of meaningful relationships

- There are lots of possible relationships. For example,
 - $V1 < V2 + K$ ($V1$ is constrained by $V2 + K$)
 - $V1 = f(V2)$, where f is any function
 - $V1$ is an enumerated variable but the set of choices for $V1$ is determined by the value of $V2$
- Relations are often reciprocal, so if $V2$ constrains $V1$, then $V1$ might constrain $V2$ (try to change $V2$ after setting $V1$)
- Given the relationship,
 - Try to enter relationship-breaking values everywhere that you can enter $V1$ and $V2$.
 - Pay attention to unusual entry options, such as editing in a display field, import, revision using a different component or program
- Once you achieve a mismatch between $V1$ and $V2$, the program's data no longer obey rules the programmer expected would be obeyed, so anything that assumes the rules hold is vulnerable. Do follow-up testing to discover serious side effects of the mismatch

Tabular Format for Data Relationships

Many relationships among data:

- Independence
 - Varying one has no effect on the value or permissible values of the other.
- Causal determination
 - By changing the value of one, we determine the value of the other.
 - For example, in MS Word, the extent of shading of an area depends on the object selected. The shading differs depending on Table vs. Paragraph.
- Constrained to a range
 - For example, the width of a line must be less than the width of the page.
 - In a date field, the permissible dates are determined by the month (and the year, if February).
- Selection of rules
 - Example, hyphenation rules depend on the language you choose.

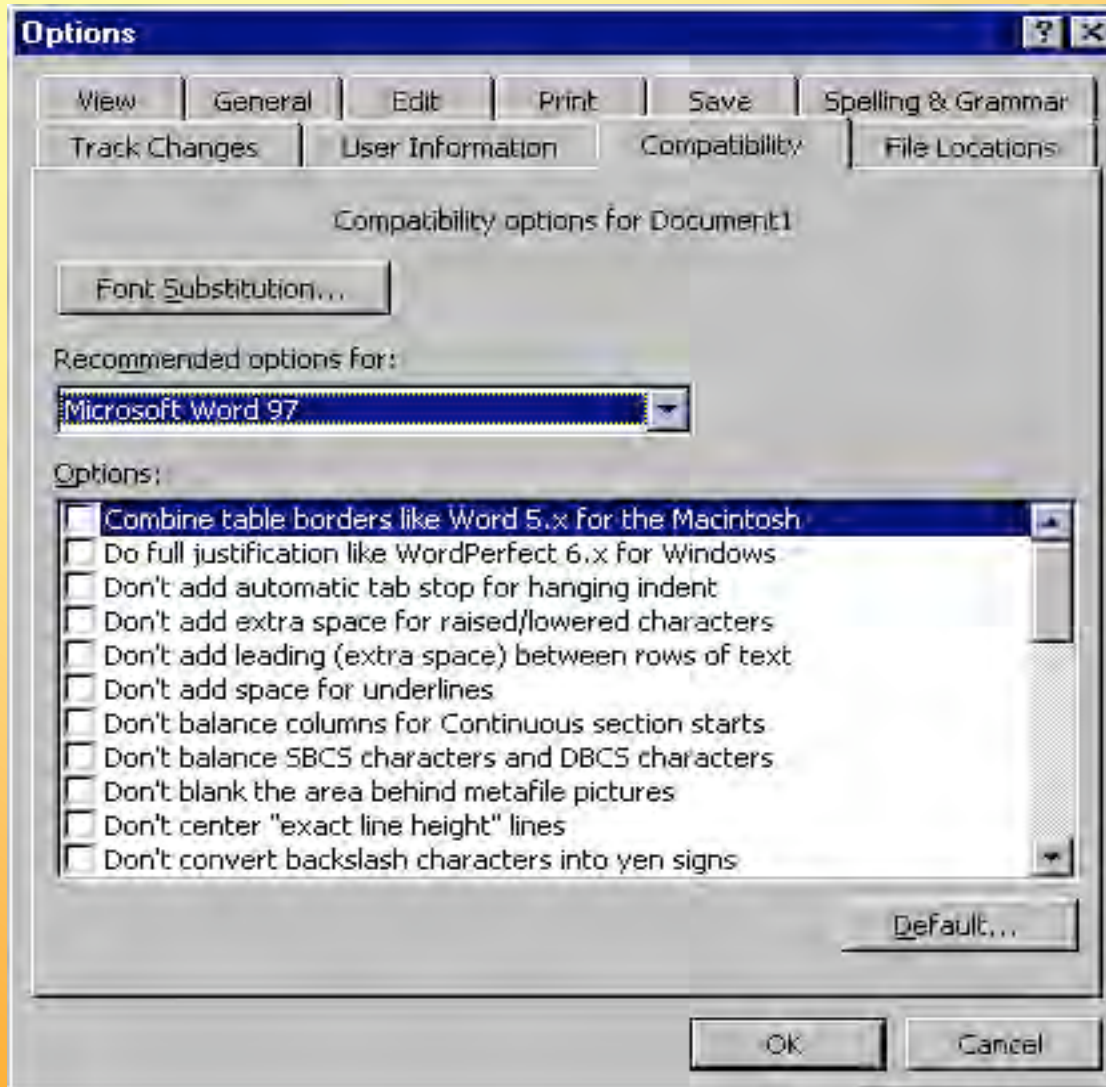
Tabular Format for Data Relationships

Many relationships among data:

- Logical selection from a list
 - processes the value you entered and then figures out what value to use for the next variable. Example: timeouts in phone dialing:
 - 0 on complete call 555-1212 but 95551212?
 - 10 on ambiguous completion, 955-5121
 - 30 seconds incomplete 555-121
- Logical selection *of* a list:
 - For example, in printer setup, choose:
 - OfficeJet, get Graphics Quality, Paper Type, and Color Options
 - LaserJet 4, get Economode, Resolution, and Half-toning.

Look at Marick (*Craft of Software Testing*) for discussion of catalogs of tests for data relationships.

Complex Data Relationships



Data Relationship Table

- Looking at the Word options, you see the real value of the data relationships table. Many of these options have a lot of repercussions (they impact many features).
- You might analyze all of the details of all of the relationships later, but for now, it is challenging just to find out what all the relationships ARE.
- The table guides exploration and will surface a lot of bugs.

PROBLEM

- Works great for *this* release. Next release, what is your support for more exploration?

Black Box Software Testing

Risk-Based Testing

Risk-based testing

- Tag line
 - Imagine a problem, then look for it
- Fundamental question or goal
 - Drive testing from a perspective of risk, rather than activities, coverage, people, or available oracle.
- Paradigmatic case(s)
 - Equivalence class analysis, reformulated
 - Organizational or project risks (generally risk-causal factors)
 - Failure Mode and Effects Analysis (FMEA)
 - Operational profiles
- Pathological cases
 - Project numerology

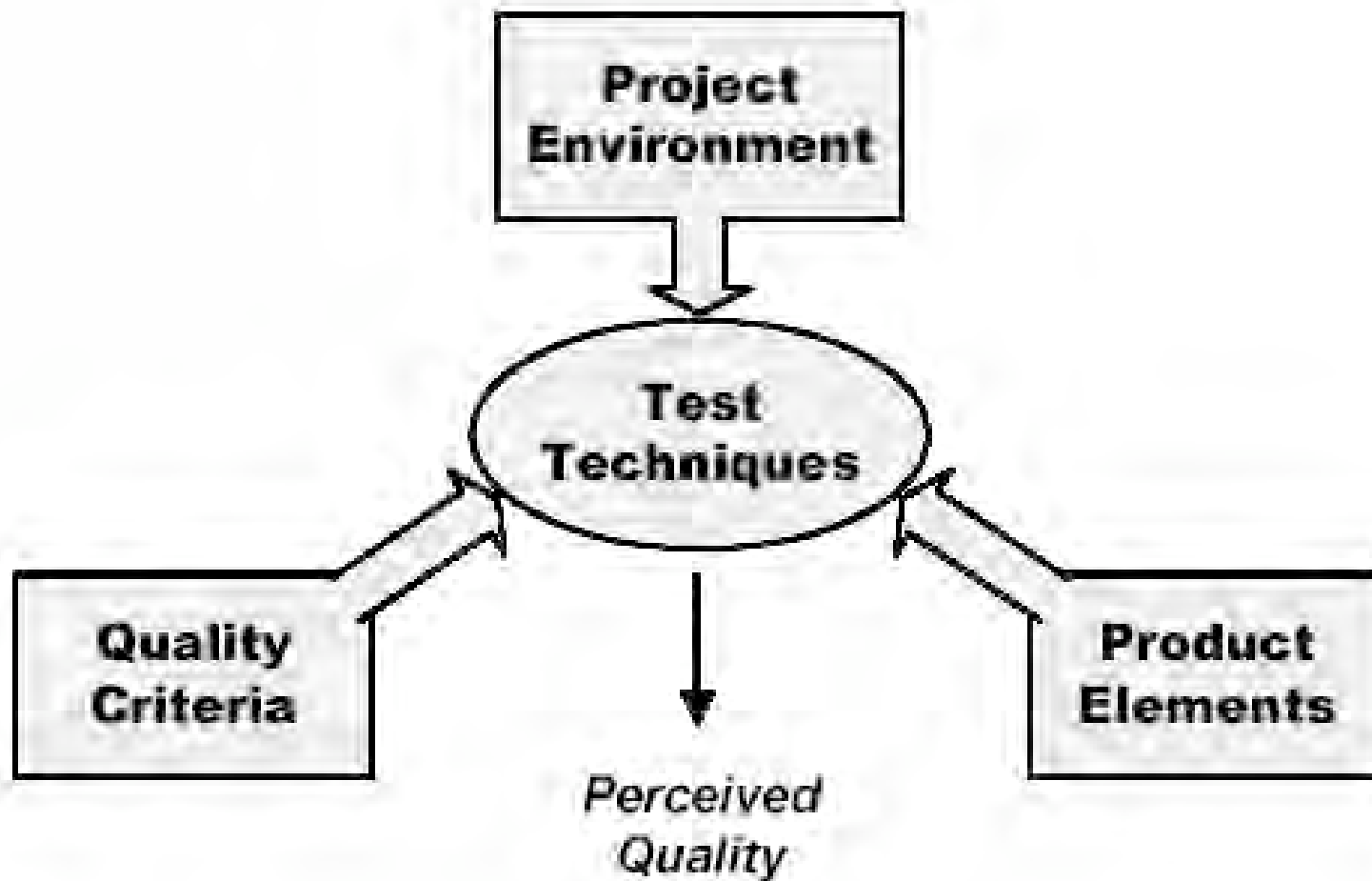
The risk-based approach to domain testing

- We studied domain testing previously, considering four different ways to think about the meaning and practice of the domain analysis. The *risk-based approach* looks like this:
 - Start by identifying a risk (a problem the program *might* have).
 - Progress by discovering a class (an equivalence class) of tests that could expose the problem.
 - Question every test candidate
 - What kind of problem do you have in mind?
 - How will this test find that problem? (Is this in the right class?)
 - What power does this test have against that kind of problem? Is there a more powerful test? A more powerful suite of tests? (Is this the best representative?)
 - Use the best representatives of the test classes to expose bugs

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.***

Bach's heuristic test strategy model: A structure for thinking about risks



Heuristic test strategy model

- The **Heuristic Test Strategy Model** is a set of patterns for designing a test strategy. The immediate purpose of this model is to remind testers of what to think about when they are creating tests. Ultimately, it is intended to be customized and used to facilitate dialog, self-directed learning, and more fully conscious testing among professional testers.
- **Quality Criteria** are high-level oracles. Use them to determine or argue that the product has problems. Quality criteria are multidimensional, often in conflict with each other.
- **Product Elements** are all the things that make up the product. They're what you intend to test. Software is so complex and invisible that you should take care to assure that you don't miss something you need to examine.
- **Project Environment** includes resources, constraints, and other forces in the project that enable us to test, while also keeping us from doing a perfect job. Make sure that you make use of the resources you have available, while respecting your constraints.
- **Test Techniques** are strategies for creating tests. All techniques involve some sort of analysis of project environment, product elements, and quality criteria.
- **Perceived Quality** is the result of testing. You can never know the "actual" quality of a software product, but through the application of a variety of tests, you can derive an informed assessment of it.

Risk-based testing

- Here how we'll focus this section
 - *Project factors*: these constrain, enable or should focus our testing
 - *Failure modes*: imagine how the product can fail, then design/use tests that can expose this type of failure. We analyze these from three angles:
 - *Product element (or, component) failures*: What are the parts of the product and, for each part, how could it fail?
 - *Operational failures*: How do things go wrong when we use the product to do real things? *Because the issues overlap so much, we currently fold this into the product element and quality attribute analyses.*
 - *Quality attribute failures*: What quality attributes (e.g. accessibility, usability, maintainability) do we most care about, or for each attribute, how could we expose inadequacies in the product?
 - *Common programming errors*: What types of mistakes are common, and for each type of error, is there a specific technique -- **an attack** -- optimized for exposing this kind of mistake?
 - *Operational profiles*: If we know the usage patterns of our customers / users, we can focus testing on the most common patterns and minimize time on things that no one will do.

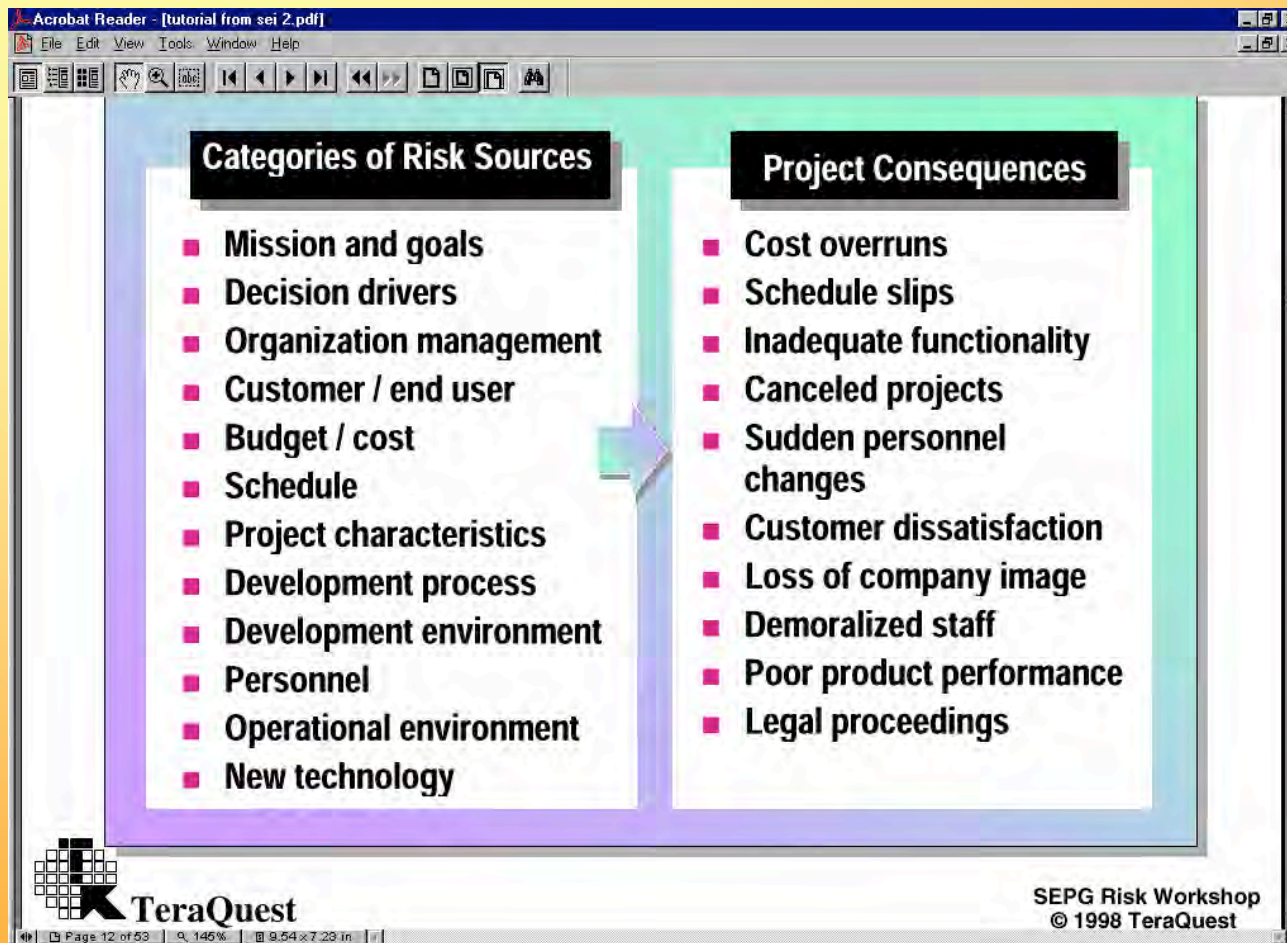
Risk-based testing



Project-Level Factors

Classic, project-level risk analysis

Project-level risk analyses usually consider risks factors that can make the project as a whole fail, and how to manage those risks.



Project-level risk analysis

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)

The problem for our purposes is that this level of analysis doesn't give us much guidance as to how to test.

Risk heuristics: Where to look for errors

Sometimes risks associated with the project as a whole or with the staff or management of the project can guide our testing.

- **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

Risk heuristics: 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

Risk heuristics: 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 the contract but yield a failed 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

Risk heuristics: 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

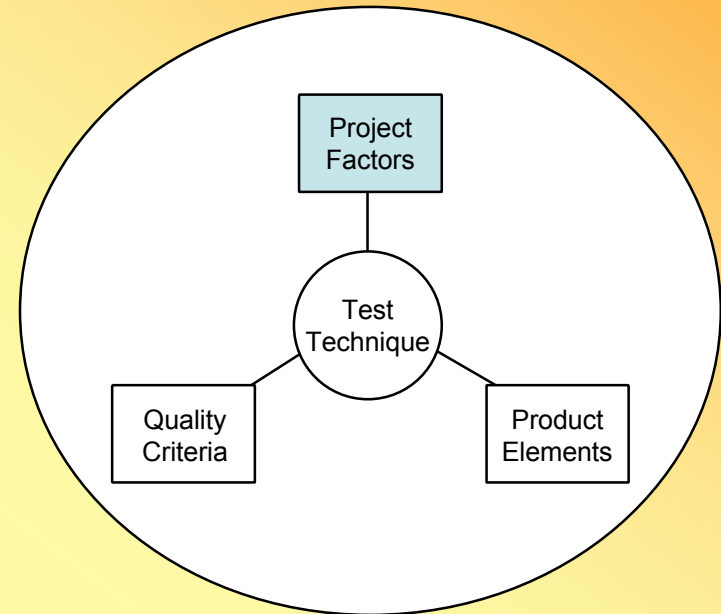
Risk heuristics: Where to look for errors

- **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

Heuristic test strategy model: Project environment factors

- **Customers.** *Anyone who is a client of the test project.*
- **Information.** *Information about the product or project that is needed for testing.*
- **Team.** *Anyone who will perform or support testing.*
- **Equipment & Tools.** *Hardware, software, or documents required to administer testing.*
- **Schedules.** *The sequence, duration, and synchronization of events.*
- **Test Items.** *The product to be tested.*
- **Deliverables.** *The observable products of the test project.*
- **Logistics and Budget.**



These aspects of the environment constrain and enable the testing project

Project environment factors

- **Customers.** Anyone who is a client of the test project.
 - Do you know who your customers are? Whose opinions matter? Who benefits or suffers from the work you do?
 - Do you have contact and communication with your customers?
 - Maybe your customers have strong ideas about what tests you should create and run.
 - Maybe they have conflicting expectations. You may have to help identify and resolve those.
 - Maybe they can help you test, in some way.

Project environment factors

- **Information.** Information about the product or project that is needed for testing.
 - Do you have all the information that you need in order to test reasonably well?
 - Do you need to familiarize yourself with the product more, before you will know how to test it?
 - Is your information current? How are you apprised of new or changing information?

Project environment factors

- **Team.** Anyone who will perform or support testing.
 - Do you know who will be testing?
 - Are there people not on the “test team” who might be able to help? People who’ve tested similar products before and might have advice? Writers? Users? Programmers?
 - Do you have enough people with the right skills to fulfill a reasonable test strategy?
 - Does the team have special skills or motivation to use particular test techniques?
 - Is any training needed? Is any available?
 - Extent to which they are focused or are multi-tasking?
 - Organization: collaboration & coordination of the staff?
 - Is there an independent test lab?

Project environment factors

- **Equipment & Tools.** Hardware, software, or documents required to administer testing.
 - *Hardware:* Do we have all the equipment you need to execute the tests? Is it set up and ready to go?
 - *Automation:* Are any test automation tools needed? Are they available?
 - *Probes / diagnostics:* Which tools needed to aid observation of the product under test?
 - *Matrices & Checklists:* Are any documents needed to track or record the progress of testing? Do any exist?

Project environment factors

- **Schedules.** The sequence, duration, and synchronization of events.
 - *Testing:* How much time do you have? Are there tests better to create later than to create them now?
 - *Development:* When will builds be available for testing, features added, code frozen, etc.?
 - *Documentation:* When will the user documentation be available for review?
 - *Hardware:* When will the hardware you need to test with (more generally, the 3rd party materials you need) be available and set up?

Project environment factors

- **Test Items.** The product to be tested.
 - *Availability:* Do you have the product to test?
 - *Volatility:* Is the product constantly changing? What will be the need for retesting?
 - *Testability:* Is the product functional and reliable enough that you can effectively test it?

Project environment factors

- **Deliverables.** The observable products of the test project.
 - *Content:* What sort of reports will you have to make? Will you share your working notes, or just the end results?
 - *Purpose:* Are your deliverables provided as part of the product? Does anyone else have to run your tests?
 - *Standards:* Is there a particular test documentation standard you're supposed to follow?
 - *Media:* How will you record and communicate your reports?

Project environment factors

- **Logistics:** Facilities and support needed for organizing and conducting the testing
 - Do we have the supplies / physical space, power, light / security systems (if needed) / procedures for getting more?
- **Budget:** Money and other resources for testing
 - Can we afford the staff, space, training, tools, supplies, etc.?

Risk-based testing



Failure Modes

Failure mode lists / Risk catalogs / Bug taxonomies

- A *failure mode* is, essentially, a way that the program could fail.
- Example: **Portion of risk catalog for installer products:**
 - Wrong files installed
 - temporary files not cleaned up
 - old files not cleaned up after upgrade
 - unneeded file installed
 - needed file not installed
 - correct file installed in the wrong place
 - Files clobbered
 - older file replaces newer file
 - user data file clobbered during upgrade
 - Other apps clobbered
 - file shared with another product is modified
 - file belonging to another product is deleted

Failure modes can guide testing

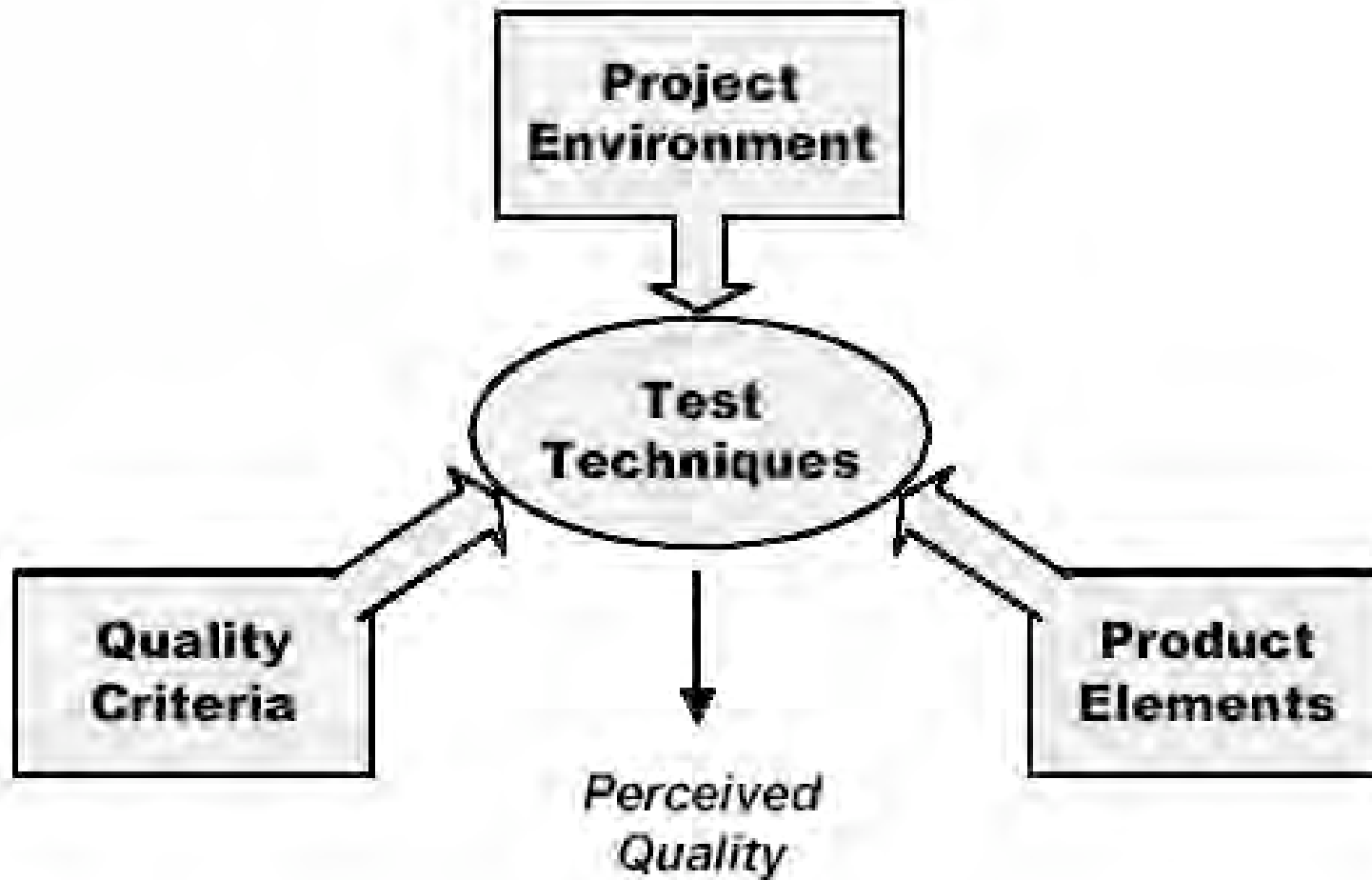
Testing Computer Software listed 480 common bugs. We used the list for:

- Test idea generation
 - 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.
- Test plan auditing
 - Pick categories to sample from
 - From each category, find a few potential defects in the list
 - For each potential defect, ask whether the software under test could have this defect
 - If it is theoretically possible that the program could have the defect, ask whether the test plan could find the bug if it was there.
- Getting unstuck
 - Look for classes of problem outside of your usual box
- Training new staff
 - Expose them to what can go wrong, challenge them to design tests that could trigger those failures

Build your own catalog of failure modes

- 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.
- There are plenty of sources to check for common failures in the common platforms
 - www.bugnet.com
 - www.cnet.com
 - trade press gossip about bugs
 - links from www.winfiles.com
 - various mailing lists

Building failure mode lists: Bach's heuristic test strategy model

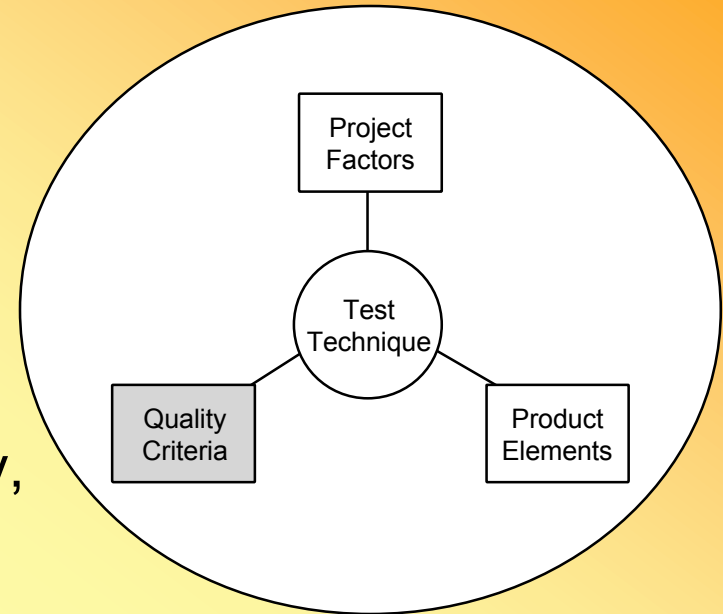


Risk-based testing: Failure modes

- Imagine how the product can fail, then design/use tests that can expose this type of failure. We analyze these from three angles:
 - ***Quality attribute failures:*** What quality attributes (e.g. accessibility, usability, maintainability) do we most care about, or for each attribute, how could we expose inadequacies in the product?
 - ***Product element (or, component) failures:*** What are the parts of the product and, for each part, how could it fail?
 - ***Operational failures:*** How do things go wrong when we use the product to do real things? *Because the issues overlap so much, we currently fold these into the product element and quality attribute analyses.*
 - ***Common programming errors:*** What types of mistakes are common, and for each type of error, is there a specific technique -- *an attack* -- optimized for exposing this kind of mistake?

Quality criteria

- Quality criteria are high-level oracles. Use them to determine or argue that the product has problems. Quality criteria are multidimensional, often in conflict with each other.
- Each quality criterion is a risk category, such as
 - “the risk of inaccessibility.”
- Failure mode listing:
 - Build a list of examples of each category.
 - Look for bugs similar to the examples.



Quality criteria categories: Operational criteria

- **Capability.** Can it perform the required functions?
- **Reliability.** Will it work well and resist failure in all required situations?
 - **Error handling:** product resists failure in response to errors, is graceful when it does fail, and recovers readily.
 - **Data Integrity:** data in the system is protected from loss or corruption.
 - **Security:** product is protected from unauthorized use.
 - **Safety:** product will not fail in such a way as to harm life or property.

Quality criteria categories: Operational criteria

- **Usability.** How easy is it for a real user to use the product?
 - **Learnability:** operation of the product can be rapidly mastered by the intended user.
 - **Operability:** product can be operated with minimum effort and fuss.
 - **Throughput:** how quickly the user can do the complete task.
 - **Accessibility:** product is usable in the face of the user's limitations or disabilities
- **Performance.** How speedy and responsive is it?
- **Concurrency:** appropriately handles multiple parallel tasks and behaves well when running in parallel with other processes.
- **Scalability:** appropriately handles increases in number of users, tasks, or attached resources.

Quality criteria categories: Operational criteria

- ***Installability and Uninstallability***. How easily can it be installed onto (and uninstalled from) its target platform?
- ***Compatibility***. How well does it work with external components & configurations?
 - ***Application Compatibility***: product works in conjunction with other software products.
 - ***Operating System Compatibility***: product works with a particular operating system.
 - ***Hardware Compatibility***: product works with particular hardware components and configurations.
 - ***Backward Compatibility***: the product works with earlier versions of itself.
 - ***Resource Usage***: the product doesn't unnecessarily hog memory, storage, or other system resources.

Quality criteria categories: Development criteria

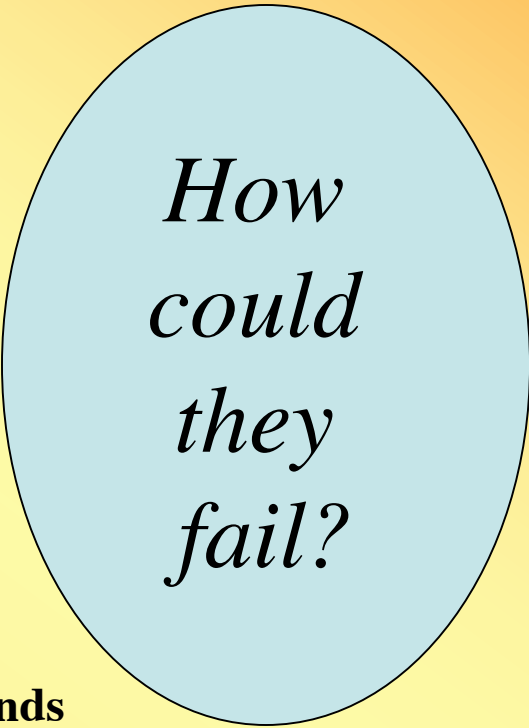
- ***Supportability***. How economical will it be to provide support to users of the product?
- ***Testability***. How effectively can the product be tested?
- ***Maintainability***. How economical is it to build, fix or enhance the product?
- ***Portability***. How economical will it be to port or reuse the technology elsewhere?
- ***Localizability***. How economical will it be to publish the product in another language?

Quality criteria categories: Development criteria

- **Conformance to Standards.** How well it meets standardized requirements, such as:
 - **Coding Standards.** Agreements among the programming staff or specified by the customer.
 - **Regulatory Standards.** Attributes of the program or development process made compulsory by legal authorities.
 - **Industry Standards.** Includes widely accepted guidelines (e.g. user interface conventions common to a particular UI) and formally adopted standards (e.g IEEE standards).

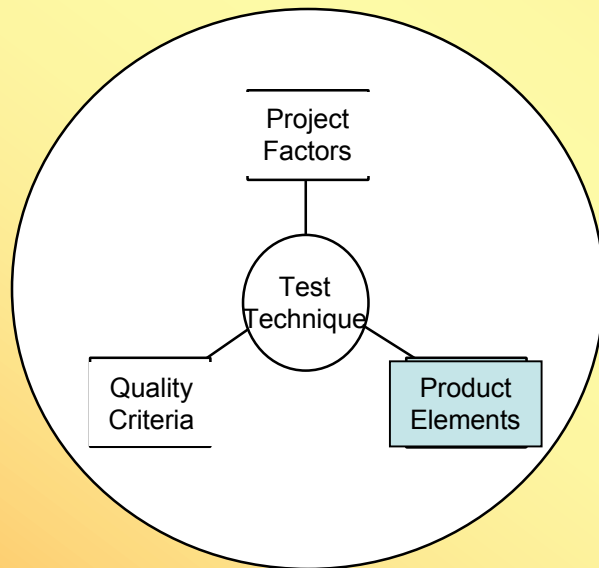
Building failure mode lists from product elements: Shopping cart example

- Think in terms of the components of your product
 - **Structures: Everything that comprises the logical or physical product**
 - Database server
 - Cache server
 - **Functions: Everything the product does**
 - Calculation
 - Navigation
 - Memory management
 - Error handling
 - **Data: Everything the product processes**
 - Human error (retailer)
 - Human error (customer)
 - **Operations: How the product will be used**
 - Upgrade
 - Order processing
 - **Platforms: Everything on which the product depends**
 - » Adapted from Giri Vijayaraghavan's Master's thesis.



*How
could
they
fail?*

Product elements: ***A product is...***



An experience or solution provided to a customer

Everything that comes in the box, plus the box!

Functions and data, executed on a platform, that serve a purpose for a user.

- 1 A software product is much more than code.
- 2 It involves a purpose, platform, and user.
- 3 It consists of many interdependent *elements*.

Product elements

- **Structures: *Everything that comprises the physical product***
 - ***Code***: the code structures that comprise the product, from executables to individual routines
 - ***Interfaces***: points of connection and communication between subsystems
 - ***Hardware***: hardware components integral to the product
 - ***Non-executable files***: any files other than programs, such as text files, sample data, help files, graphics, movies, etc.
 - ***Ephemera and collaterals***: anything beyond software and hardware, such as paper documents, web links and content, packaging, license agreements, etc.

Product elements

- **Functions: *Everything the product does.***
 - ***User Interface:*** functions that mediate the exchange of data with the user
 - ***System Interface:*** functions that exchange data with something other than the user, such as with other programs, hard disk, network, printer, etc.
 - ***Application:*** functions that define or distinguish the product or fulfil core requirements.
 - ***Multimedia:*** self-executing or automatically executing sounds, bitmaps, movies embedded in the product.
 - ***Error Handling:*** functions that detect and recover from errors, including all error messages.
 - ***Interactions:*** interactions or interfaces between functions within the product.
 - ***Testability:*** functions provided to help test the product, such as diagnostics, log files, asserts, test menus, etc.

Product elements

- **Data: *Everything the product processes***
 - ***Input***: data that is processed by the product
 - ***Output***: data that results from processing by the product
 - ***Preset***: data supplied as part of the product or otherwise built into it, such as prefabricated databases, default values, etc.
 - ***Persistent***: data stored internally and expected to persist over multiple operations. This includes modes or states of the product, such as options settings, view modes, contents of documents, etc.
 - ***Temporal***: any relationship between data and time, such as the number of keystrokes per second, date stamps on files, or synchronization of distributed systems.

Product elements

- **Platform: *Everything on which the product depends***
 - ***External Hardware***: components and configurations that are not part of the shipping product, but are required (or optional) in order for the product to work. Includes CPU's, memory, keyboards, peripheral boards, etc.
 - ***External Software***: software components and configurations that are not a part of the shipping product, but are required (or optional) in order for the product to work. Includes operating systems, concurrently executing applications, drivers, fonts, etc.

Product elements

- **Operations: *How the product will be used***

- *Users*. attributes of the various kinds of users.
- *Usage Profile*: the pattern of usage, over time, including patterns of data that the product will typically process in the field. This varies by user and type of user.
- *Environment*: the physical environment in which the product will be operated, including such elements as light, noise, and distractions.

Product elements: Coverage

Product coverage is the proportion of the product that has been tested.

- **There are as many kinds of coverage as there are ways to model the product.**

- Structural
- Functional
- Temporal
- Data
- Platform
- Operations

See Software Negligence & Testing Coverage for 101 examples of coverage “measures” and Measurement of the Extent of Testing for a broader discussion of measurement theory and coverage, both at www.kaner.com

Programming errors and attacks



Some errors are so common that there are well-known attacks for them.

An *attack* is a stereotyped class of tests, optimized around a specific type of error.

Think back to domain testing:

- ***Boundary testing for numeric input fields is an example of an attack. The error is mis-specification (or mis-typing) of the upper or lower bound of the numeric input field.***

Programming errors and attacks

- James Whittaker and Alan Jorgensen pulled together a powerful collection of attacks (and examples).
- In his book, *How to Break Software*, Professor Whittaker expanded the list and, for each attack, discussed
 - When to apply it
 - What software errors make the attack successful
 - How to determine if the attack exposed a failure
 - How to conduct the attack, and
 - An example of the attack.
- We'll list *How to Break Software's* attacks here, but recommend the book's full discussion.
- Following that, we'll describe a few additional attacks from an exploratory testers' attack list put together at the Los Altos Workshop in Software Testing, July 1999.

Programming errors and attacks

- User interface attacks: Exploring the input domain
 - Attack 1: Apply inputs that force all the error messages to occur
 - Attack 2: Apply inputs that force the software to establish default values
 - Attack 3: Explore allowable character sets and data types
 - Attack 4: Overflow input buffers
 - Attack 5: Find inputs that may interact and test combinations of their values
 - Attack 6: Repeat the same input or series of inputs numerous times

» From Whittaker, *How to Break Software*

Programming errors and attacks

- User interface attacks: Exploring outputs
 - Attack 7: Force different outputs to be generated for each input
 - Attack 8: Force invalid outputs to be generated
 - Attack 9: Force properties of an output to change
 - Attack 10: Force the screen to refresh.

» From Whittaker, *How to Break Software*

Programming errors and attacks

Testing from the user interface: Data and computation

- Exploring stored data
 - Attack 11: Apply inputs using a variety of initial conditions
 - Attack 12: Force a data structure to store too many or too few values
 - Attack 13: Investigate alternate ways to modify internal data constraints

» From Whittaker, *How to Break Software*

Programming errors and attacks

Testing from the user interface: Data and computation

- Exploring computation and feature interaction
 - Attack 14: Experiment with invalid operand and operator combinations
 - Attack 15: Force a function to call itself recursively
 - Attack 16: Force computation results to be too large or too small
 - Attack 17: Find features that share data or interact poorly

» From Whittaker, *How to Break Software*

Programming errors and attacks

System interface attacks

- Testing from the file system interface: Media-based attacks
 - Attack 1: Fill the file system to its capacity
 - Attack 2: Force the media to be busy or unavailable
 - Attack 3: Damage the media
- Testing from the file system interface: File-based attacks
 - Attack 4: Assign an invalid file name
 - Attack 5: Vary file access permissions
 - Attack 6: Vary or corrupt file contents

» From Whittaker, *How to Break Software*

Programming errors and attacks



We pulled together a variety of common attacks at LAWST 7 (Los Altos Workshop on Software Testing) in 1999.

Some of these overlap with Whittaker's attacks.

The following slides briefly describe some of the other LAWST 7 attacks, or provide a different approach to some of the attacks described in *How to Break Software*.

Many of the ideas in these notes were reviewed and extended by 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. We appreciate their contributions.

Programming errors and attacks

Follow up recent changes

- Code changes cause side effects
 - Test the modified feature / change itself.
 - Test features that interact with this one.
 - Test data that are related to this feature or data set.
 - Test scenarios that use this feature in complex ways.

Programming errors and attacks

Explore 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

Programming errors and attacks

Explore data relationships

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

For more discussion of this table, see Combination Testing

Programming errors and attacks

Explore data relationships

- There are lots of possible relationships. For example,
 - $V1 < V2 + K$ ($V1$ is constrained by $V2 + K$)
 - $V1 = f(V2)$, where f is any function
 - $V1$ is an enumerated variable but the set of choices for $V1$ is determined by the value of $V2$
- Relations are often reciprocal, so if $V2$ constrains $V1$, then $V1$ might constrain $V2$ (try to change $V2$ after setting $V1$)
- Given the relationship,
 - Try to enter relationship-breaking values everywhere that you can enter $V1$ and $V2$.
 - Pay attention to unusual entry options, such as editing in a display field, import, revision using a different component or program
- Once you achieve a mismatch between $V1$ and $V2$, the program's data no longer obey rules the programmer expected would be obeyed, so anything that assumes the rules hold is vulnerable. Do follow-up testing to discover serious side effects of the mismatch

Programming errors and attacks

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.

Programming errors and attacks

Interference testing: 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 (e.g. set another program's (or this program's) time-reminder to go off during the task under test)

Programming errors and attacks

Interference testing:

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

Programming errors and attacks

Interference testing: Cancel

- 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

Programming errors and attacks

Interference testing: 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.

Programming errors and attacks

Interference testing: Swap (out of memory)

- Swap the process out of memory while it's running (e.g. change focus to another application; 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.

Programming errors and attacks

Interference testing: 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*

Programming errors and attacks

Testing Error Handling

- This is Whittaker's Attack #1, but here are a few additional thoughts.
- Here are examples of the types of errors you should generate:
 - 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.)

Risk-based testing



Operational Profiles

Operational profiles

- John Musa, *Software Reliability Engineering*
 - Based on enormous database of actual customer usage data
 - Accurate information on feature use and feature interaction
 - Given the data, drive reliability by testing in order of feature use
 - Problem: low probability, high severity bugs
- **Warning, Warning Will Robinson, Danger Danger!**
 - Unverified estimates of feature use / feature interaction
 - Drive testing in order of perceived use
 - Rationalize not-fixing in order of perceived use
 - Estimate reliability in terms of bugs not found

Risk-based testing

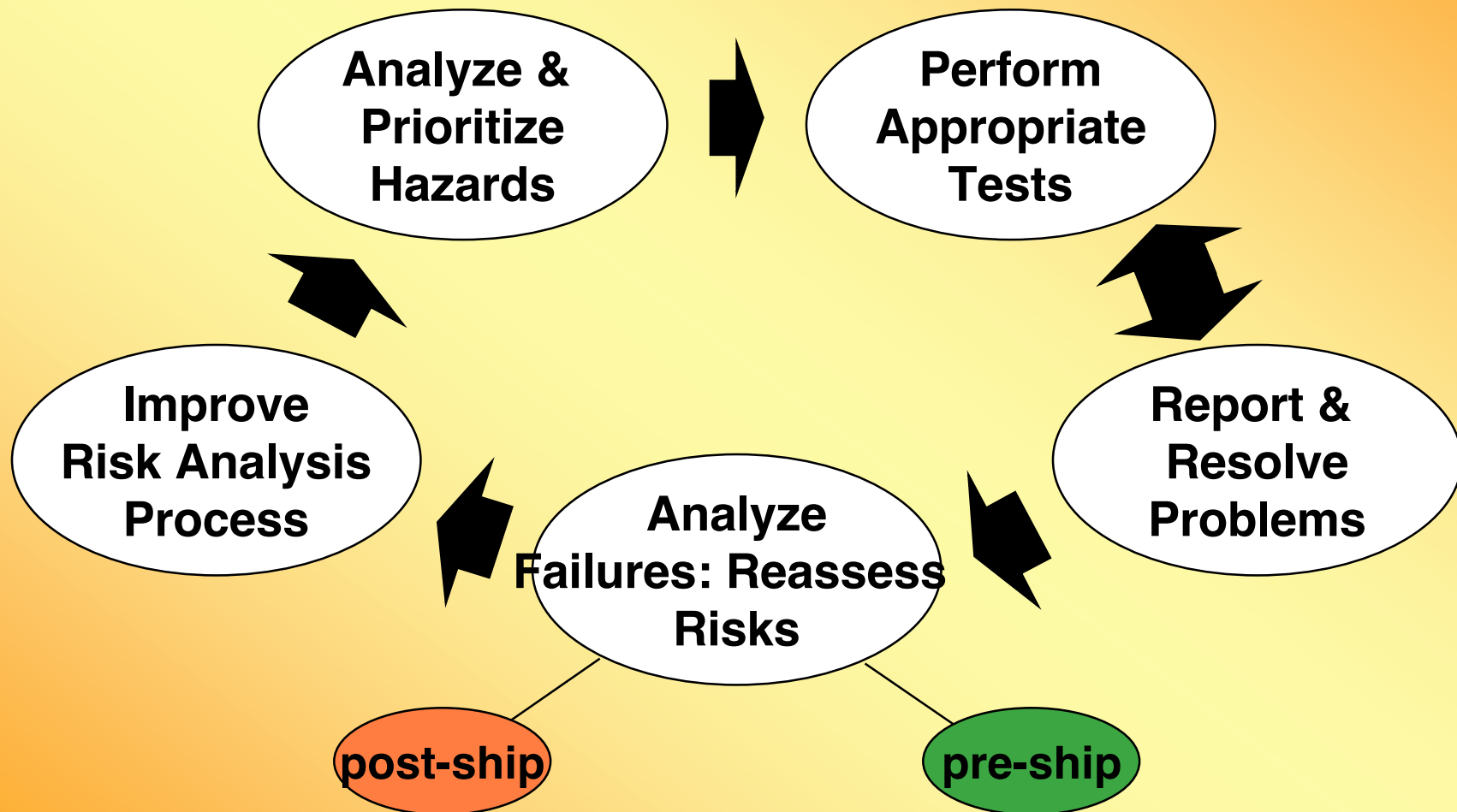


Structuring and Managing Risk-Based Testing

Here's one way to do RBT

1. Get risk ideas:
 - From risk factors or from failure mode categories (use guideword heuristics)
2. For each important idea, *determine test activities, prepare tests [that have power against that idea], shift resources* to gather information it. Otherwise, *escalate it*.
3. Maintain traceability between risks and tests.
4. Monitor and report the status of the risks as the project goes on and you learn more about them.
5. Assess coverage of the testing effort program, given a set of risk-based tests. Find holes in the testing effort.
6. If a risk is determined to be small enough, then stop testing against it.
7. On retesting an area, evaluate your tests experiences so far to determine what risks they were testing for and whether more powerful variants can be created.
8. Do at least *some* non-risk-based testing, to cover yourself in case your risk analysis is wrong.
9. Build lists of bug histories, configuration problems, tech support requests and obvious customer confusions -- deepen your lists of failure modes

Risk-driven testing cycle



Risk-based testing



Project Numerology

Some people call this risk-based test management or even risk-based testing

- 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
- Multiply estimated probability by estimated severity, this (allegedly) estimates risk
- Prioritize testing the areas in order of the estimated risk

Numerology?

- For a significant area of the product, what is the probability of error? (100%)
- What is the severity of this error? (Which error?)
- What is the difference between a level 3 and a level 4 and a level 5? (We don't and can't know, they are ordinally scaled.)
- What is the sum of two ordinal numbers? (undefined)
- What is the product of two ordinal numbers (undefined)
- What is the meaning of the risk of 12 versus 16? (none)

How “risk magnitude” can hide the right answer

- A very severe event (impact = 10)...
- that happens rarely (likelihood = 1)...
- has less magnitude than 73% of the risk space.
- A 5 X 5 risk has a magnitude that is %150 greater.
- What about Borland’s Turbo C++ project file corruption bug?
 - It cost hundreds of thousands of dollars and motivated a product recall. Yet it was a “rare” occurrence, by any stretch of the imagination. It would have scored a 10, even though it turned out to be the biggest problem in that release.

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

Black Box Software Testing

Scenario Testing

Scenario testing

- Tag line
 - Tell a persuasive story
- 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:
 - The test is *based on a story* about how the program is used, including information about the motivations of the people involved.
 - The story is *motivating*. A stakeholder with influence would push to fix a program that failed this test.
 - The story is *credible*. It not only *could* happen in the real world; stakeholders would believe that something like it probably *will* happen.
 - The story involves a *complex use* of the program *or a complex environment or a complex set of data*.
 - The test results are *easy to evaluate*. This is valuable for all tests, but is especially important for scenarios because they are complex.

Why use scenario tests?

- Learn the product
- Connect testing to documented requirements
- Expose failures to deliver desired benefits
- Explore expert use of the program
- Make a bug report more motivating
- Bring requirements-related issues to the surface, which might involve reopening old requirements discussions (with new data) or surfacing not-yet-identified requirements.

Scenarios

- Designing scenario tests is much like doing a requirements analysis, but is not requirements analysis. They rely on similar information but use it differently.
 - The requirements analyst tries to foster agreement about the system to be built. The tester exploits disagreements to predict problems with the system.
 - The tester doesn't have to reach conclusions or make recommendations about how the product should work. Her task is to expose credible concerns to the stakeholders.
 - The tester doesn't have to make the product design tradeoffs. She exposes the consequences of those tradeoffs, especially unanticipated or more serious consequences than expected.
 - The tester doesn't have to respect prior agreements. (Caution: testers who belabor the wrong issues lose credibility.)
 - The scenario tester's work need not be exhaustive, just useful.

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 do 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?
- **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?

Twelve ways to create good scenarios

1. Write life histories for objects in the system.
2. List possible users, analyze their interests and objectives.
3. Consider disfavored users: how do they want to abuse your system?
4. List system events. How does the system handle them?
5. List special events. What accommodations does the system make for these?
6. List benefits and create end-to-end tasks to check them.
7. Interview users about famous challenges and failures of the old system.
8. Work alongside users to see how they work and what they do.
9. Read about what systems like this are supposed to do.
10. Study complaints about the predecessor to this system or its competitors.
11. Create a mock business. Treat it as real and process its data.
12. Try converting real-life data from a competing or predecessor application.

Soap operas

Ashley hears about Jack's deposit when he thought he had to go. Victoria lectures her father about what's wrong with him and Nikki but Victor advises her that it's none of her business Olivia learns Dru has no regrets about leaving and takes great satisfaction in having Lily as her companion. Dru then asks Olivia why she is raking Malcolm over the coals. Stopping by Gina's, Nikki spots Brad and sits with him, admitting she doesn't want to be alone tonight. Victor stops by Mack's party at the Crimson Lights. Ashley takes a home pregnancy test. Worried about Billy, Raul makes a call and J.T. claims he doesn't know where Billy is. Raul rushes over and finds Billy out cold in the snow. Raul worries when he can't find a pulse. . . .



From a talk by Hans Buwalda

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.

Examples of story lines when used for testing

Pension fund

William starts as a metal worker for Industrial Entropy Incorporated in 1955. During his career he becomes ill, works part time, marries, divorces, marries again, gets 3 children, one of which dies, then his wife dies and he marries again and gets 2 more children....

World wide transaction system for an international bank

A fish trade company in Japan makes a payment to a vendor on Iceland. It should have been a payment in Icelandic Kronur, but it was done in Yen instead. The error is discovered after 9 days and the payment is revised and corrected, however, the interest calculation (value dating)...

From a talk by Hans Buwalda

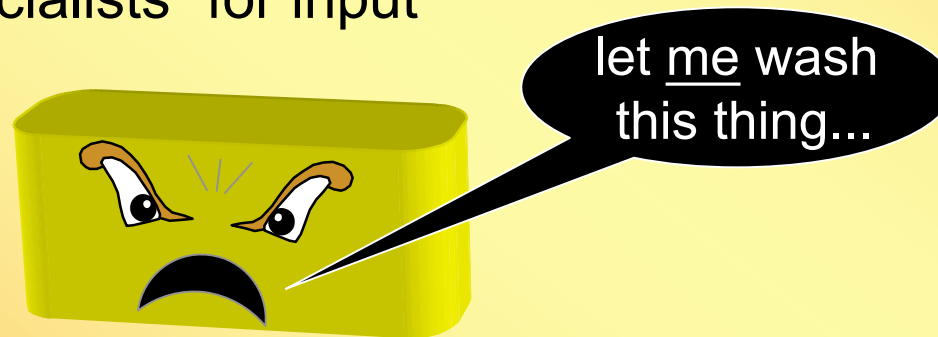
Soap operas (in testing) are not necessarily:

- “Extreme”
- Far fetched
- Long and elaborate
- Pieces of art and creativity

From a talk by Hans Buwalda

“Killer Soaps”

- More specifically aimed at finding hidden problems
- Run when everything else has passed
- One option: put a killer soap at the end of a normal cluster
- Ask the “specialists” for input



From a talk by Hans Buwalda

Risks of scenario testing

- Other approaches are better for testing early, unstable code.
 - A scenario is complex, involving many features. If the first feature is broken, the rest of the test can't be run. Once that feature is fixed, the next broken feature blocks the test.
 - Test each feature in isolation before testing scenarios, to efficiently expose problems as soon as they appear.
- Scenario tests are not designed for coverage of the program.
 - It takes exceptional care to cover all features or requirements in a set of scenario tests. Statement coverage simply isn't achieved this way.
- Reusing scenarios may lack power and be inefficient
 - Documenting and reusing scenarios seems efficient because it takes work to create a good scenario.
 - Scenarios often expose design errors but we soon learn what a test teaches about the design.
 - Scenarios expose coding errors because they combine many features and much data. To cover more combinations, we need new tests.
 - Do regression testing with single-feature tests or unit tests, not scenarios.

Black Box Software Testing

Test Design

Suggested Reading:

Kaner, Bach & Pettichord, *Testing Techniques*
in *Lessons Learned in Software Testing*.

Whittaker, *Software Testing*

Test design: Let's take stock

We've studied

- Domain testing
 - Reduce the number of test cases
 - Focus on variables (input / output / intermediate / control)
 - Pick high-power tests
- Risk-based testing
 - Identify potential failures
 - Develop powerful tests to expose errors
- Scenario testing
 - Credible, motivating stories
 - Complex combinations that can model experienced use

Test design



***Test design is
a multi-dimensional
challenge***

Test design

- What is the difference between
 - User testing?
 - Usability testing?
 - User interface testing?

Test design

Testing combines techniques that focus on:

- Testers: *who* does the testing.
- Coverage: *what* gets tested.
- Potential problems: *why* you're testing (what risk you're testing for).
- Activities: *how* you test.
- Evaluation: *how to tell whether the test passed or failed.*

*All testing involves
all five dimensions.*

Test design

- A technique focuses your attention on one or a few dimensions, leaving the others open to your judgment. You can combine a technique focused on one dimension with techniques focused on the other dimensions to achieve the result you want.
- For example:
 - Domain testing
 - Coverage (*test every variable*)
 - Potential problems (*identify risks for each variable*)
 - Risk-based testing
 - Potential problems
 - Scenario testing
 - Activity (*how you test*)

-- See *Lessons Learned*, ch. 3

What's a test case?

- Focus on procedure?
 - “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.” (IEEE)
- Focus on the test idea?
 - “A test idea is a brief statement of something that should be tested. For example, if you're testing a square root function, one idea for a test would be ‘test a number less than zero’. The idea is to check if the code handles an error case.” (Marick)

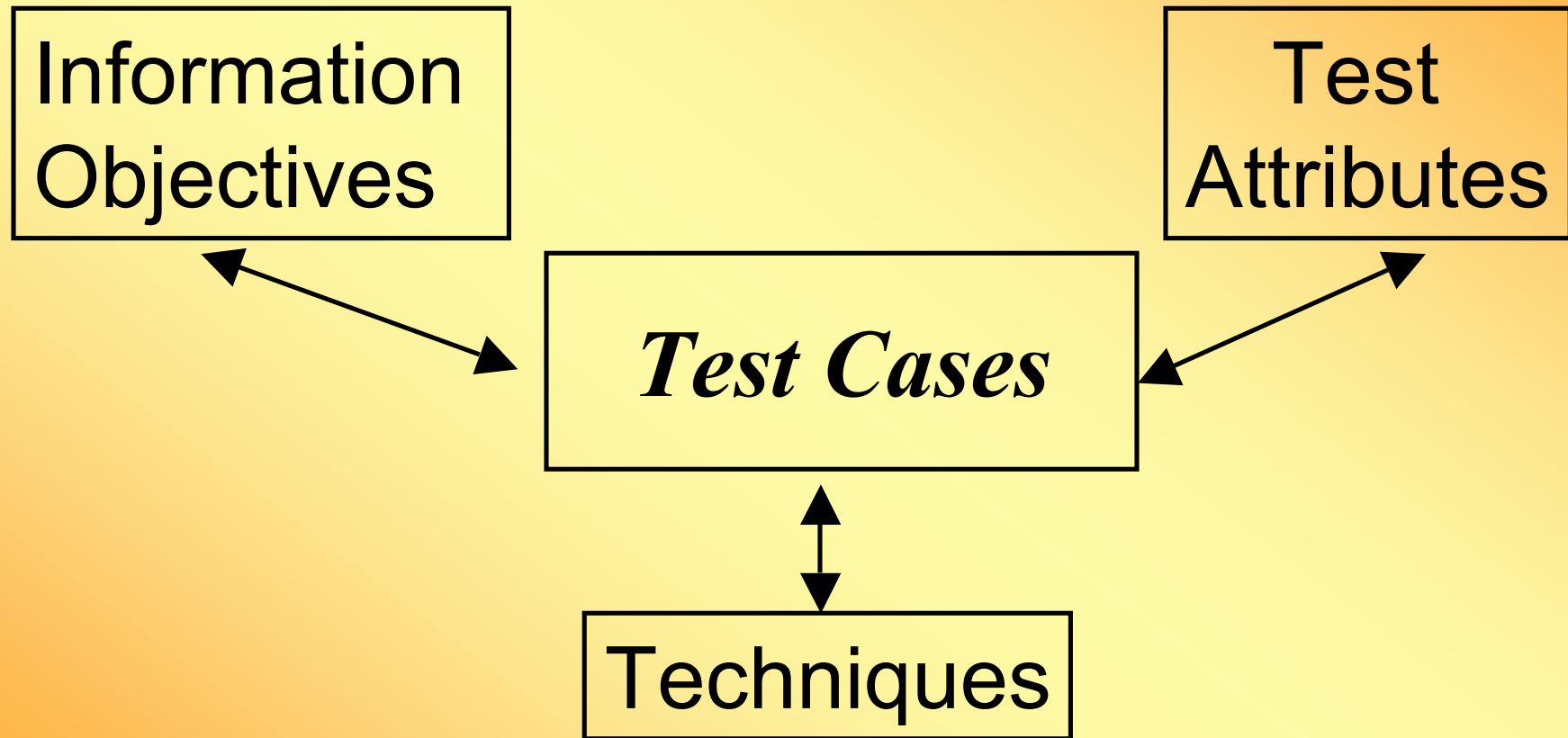
Test cases

- In my view,

**A test case is a question
you ask of the program.**

- The point of running the test is to gain information, for example whether the program will pass or fail the test.
- Implications of this approach:
 - The test must be CAPABLE of revealing valuable information
 - The SCOPE of a test changes over time, because the information value of tests changes as the program matures
 - The METRICS that count test cases are essentially meaningless because test cases merge or are abandoned as their information value diminishes.

Factors involved in test case quality



Test design

Information Objectives?

Let's try an exercise ...

Information Objectives: Which Group is Better?

Testing Group 1

	Found pre-release
Function A	100
Function B	0
Function C	0
Function D	0
Function E	0
Total	100

Testing Group 2

Function A	50
Function B	6
Function C	6
Function D	6
Function E	6
Total	74

From Marick,
Classic Testing Mistakes

Two groups test the same program.

- The functions are equally important
- The bugs are equally significant

This is artificial, but it sets up a simple context for a discussion of tradeoffs.

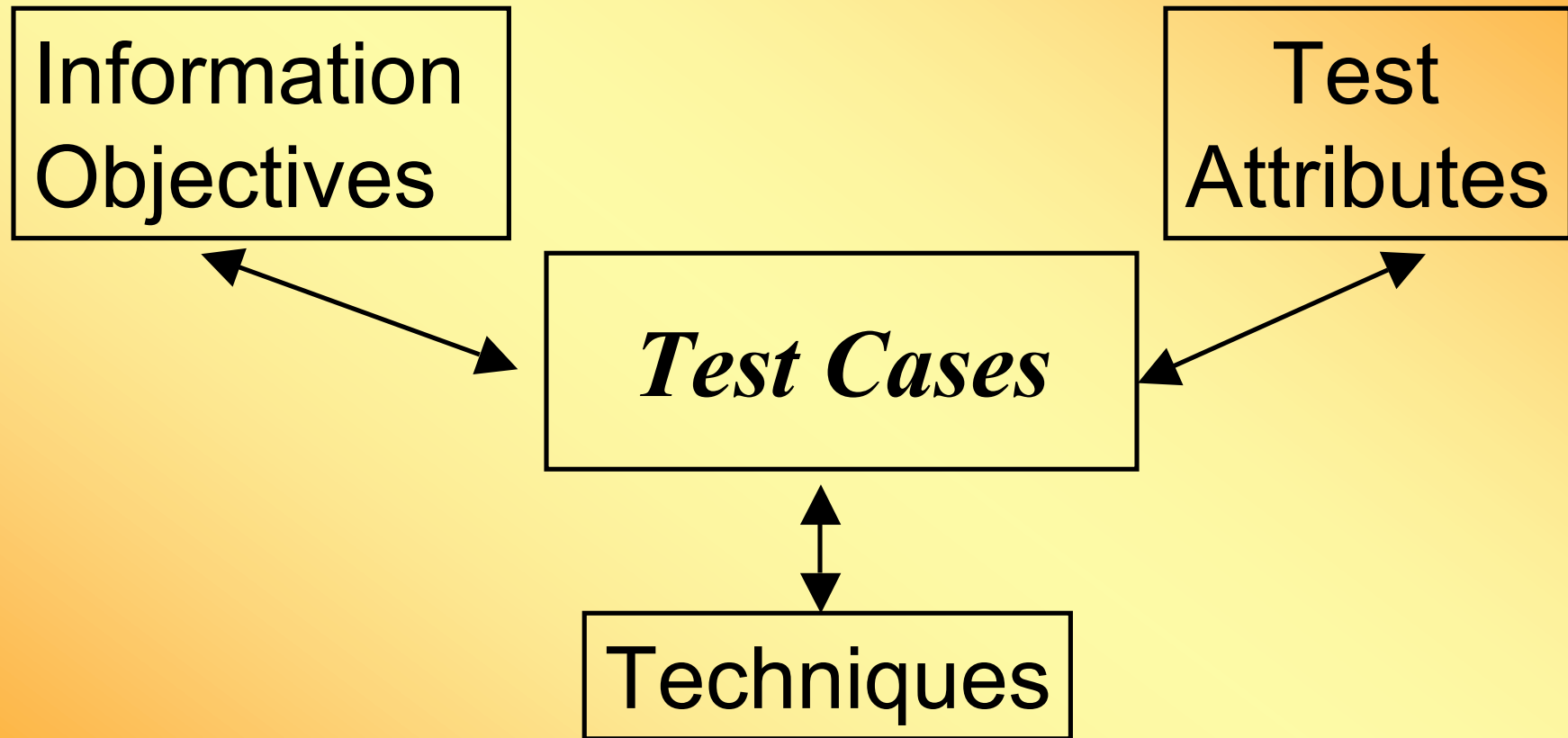
Which group is better?

	Found pre-release	Found later	Total
Function A	100	0	100
Function B	0	12	12
Function C	0	12	12
Function D	0	12	12
Function E	0	12	12
Total	100	48	148
Function A	50	50	100
Function B	6	6	12
Function C	6	6	12
Function D	6	6	12
Function E	6	6	12
Total	74	74	148

Information objectives

- Find defects
- Maximize bug count
- Block premature product releases
- Help managers make ship / no-ship decisions
- Minimize technical support costs
- Assess conformance to specification
- Conform to regulations
- Minimize safety-related lawsuit risk
- Find safe scenarios for use of the product
- Assess quality
- Verify correctness of the product
- Assure quality

Factors involved in test case quality



Test attributes

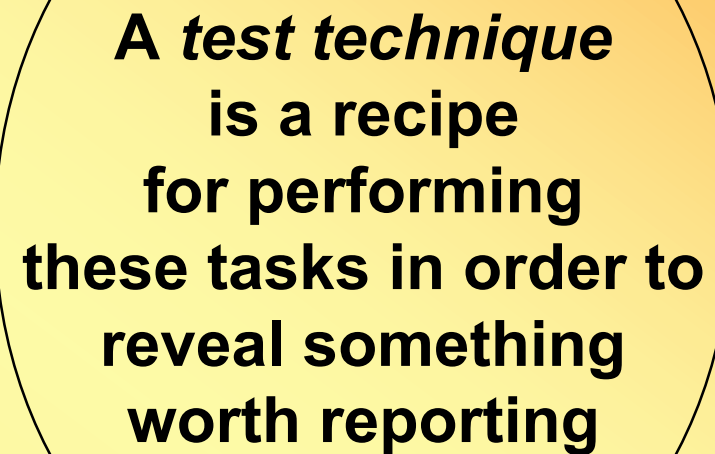
To different degrees, good tests have these attributes:

- **Power.** When a problem exists, the test will reveal it.
- **Valid.** When the test reveals a problem, it is a genuine problem.
- **Value.** It reveals things your clients want to know about the product or project.
- **Credible.** Your client will believe that people will do the things that are done in this test.
- **Representative** of events most likely to be encountered by the user. (xref. Musa's *Software Reliability Engineering*).
- **Motivating.** Your client will want to fix the problem exposed by this test.
- **Performable.** It can be performed as designed.
- **Maintainable.** Easy to revise in the face of product changes.
- **Repeatable.** It is easy and inexpensive to reuse the test.
- **Pop.** (*short for Karl Popper*) It reveal things about our basic or critical assumptions.
- **Coverage.** It exercises the product in a way that isn't already taken care of by other tests.
- **Easy to evaluate.**
- **Supports troubleshooting.** Provides useful information for the debugging programmer.
- **Appropriately complex.** As the program gets more stable, you can hit it with more complex tests and more closely simulate use by experienced users.
- **Accountable.** You can explain, justify, and prove you ran it.
- **Cost.** This includes time and effort, as well as direct costs.
- **Opportunity Cost.** Developing and performing this test may prevent you from doing other tests (or other work).

» See Kaner, *What IS a Good Test Case?*

Test Techniques

- Analyze the situation.
- Model the test space.
- Select what to cover.
- Determine test oracles.
- Configure the test system.
- Operate the test system.
- Observe the test system.
- Evaluate the test results.



***A test technique
is a recipe
for performing
these tasks in order to
reveal something
worth reporting***

Eleven dominating techniques

This list reflects our observations in the field. It is not exhaustive. We put a technique 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.

- Domain testing
- Risk-based testing
- Scenario testing
- Function testing
- Specification-based testing
- Regression testing
- Stress testing
- User testing
- State-model based testing
- High volume automated testing
- Exploratory testing

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

- Exploratory testing is not a testing technique. It's a way of thinking about testing.

- Any technique can be used in any exploratory way.
- Any competent tester does *some* exploratory testing.

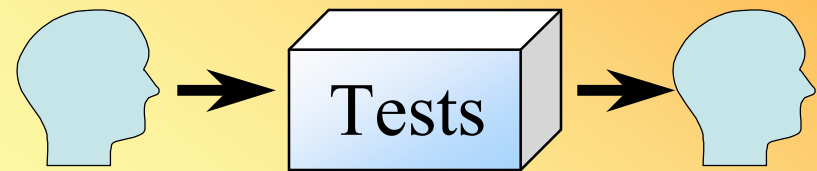
- Example -- bug regression

Report a bug, the programmer claims she fixed the bug, so you test the fix.

- Start by reproducing the steps you used in the bug report to expose the failure.
- Then vary your testing to search for side effects.
 - » These variations are not predesigned. This is an example of chartered exploratory testing.

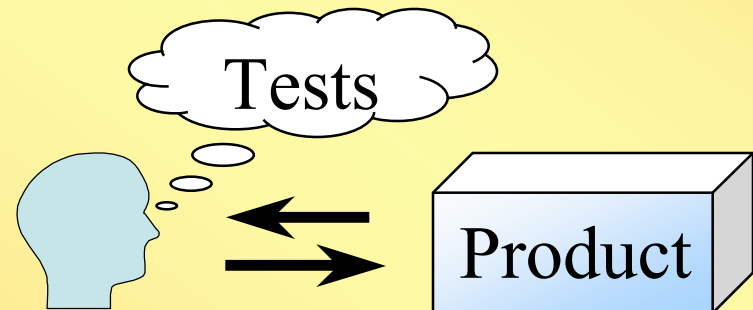
Contrasting Approaches

In *scripted* testing, tests are first designed and recorded. Then they may be executed at some later time or by a different tester.



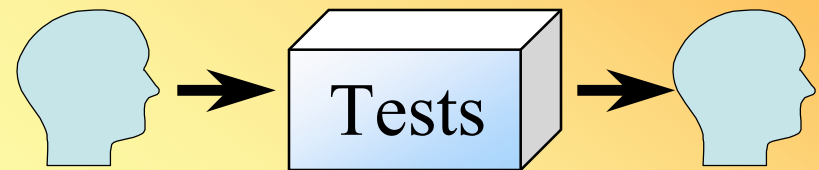
vs.

In *exploratory* testing, tests are designed and executed at the same time, and they often are not recorded.



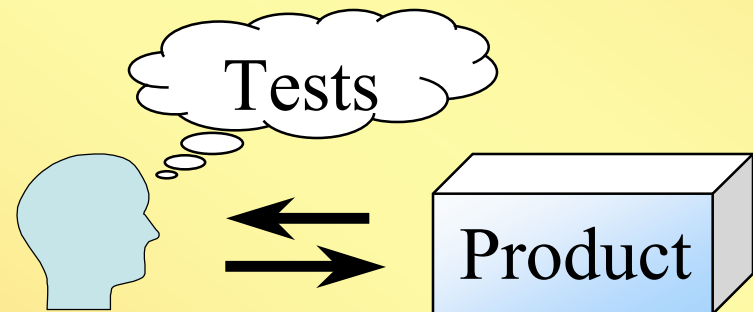
Contrasting Approaches

Scripted testing emphasizes
accountability and *decidability*.



vs.

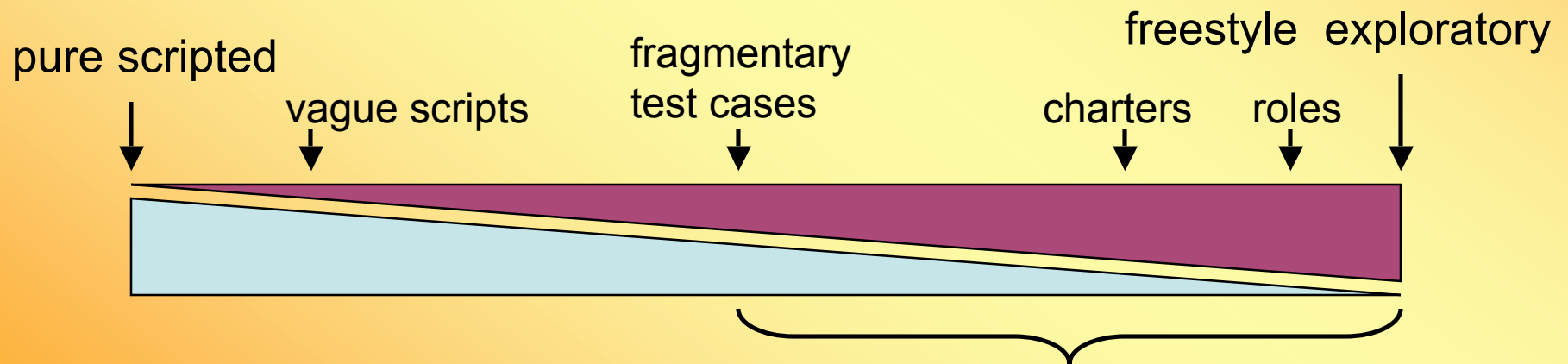
Exploratory testing emphasizes
adaptability and *learning*.



Exploratory Testing Defined

Definition


Exploratory testing is simultaneous *learning, test design, and test execution.*





When I say “exploratory testing” and don’t qualify it, I mean anything on the exploratory side of this continuum.

Exploratory Testing Tasks

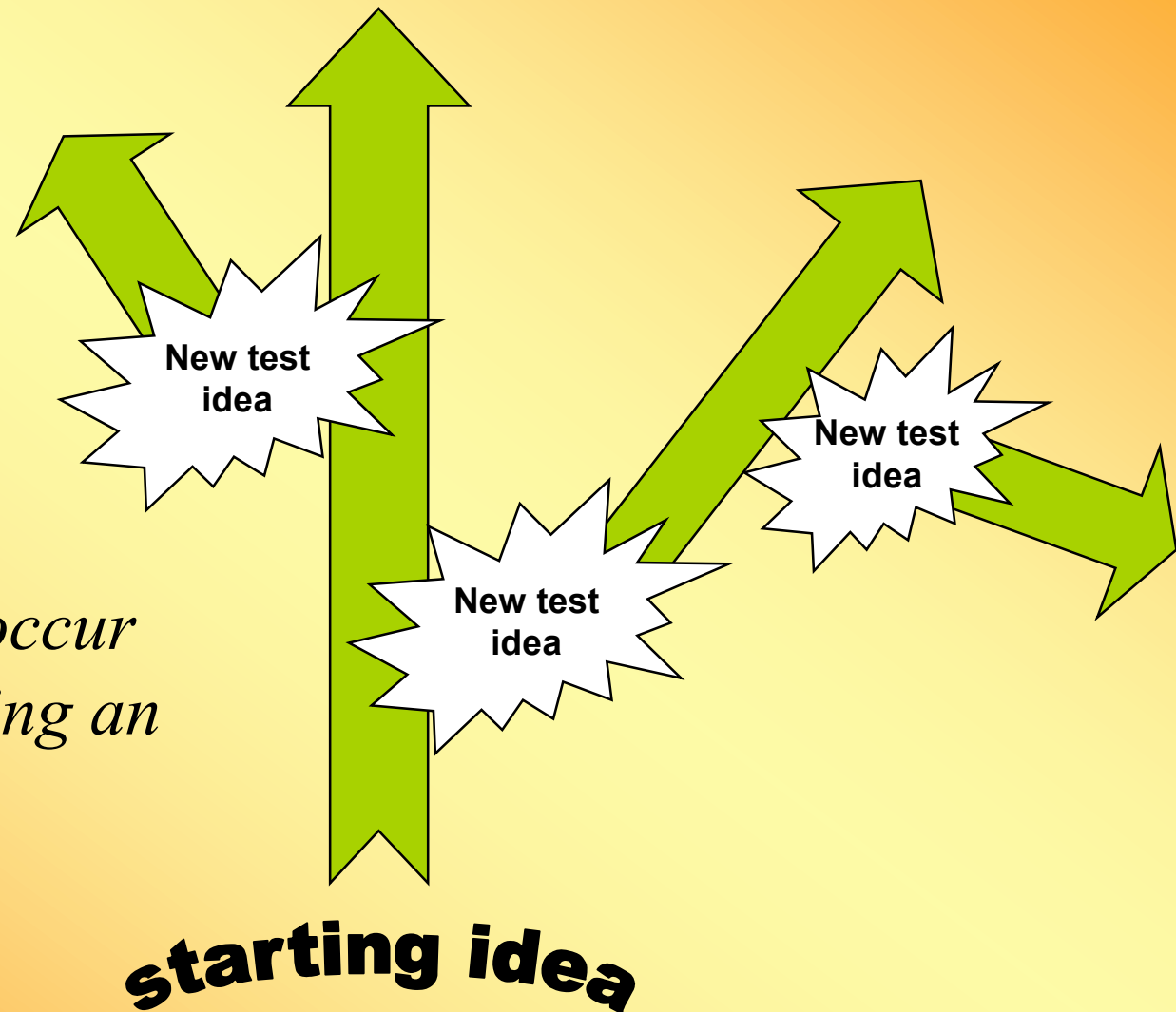
	Learning	Design Tests	Execute Tests
Product <i>(coverage)</i>	Discover the elements of the product.	Decide which elements to test.	Observe product behavior.
Quality <i>(oracles)</i>	Discover how the product should work.	Speculate about possible quality problems.	Evaluate behavior against expectations.
Techniques	Discover test design techniques that can be used.	Select & apply test design techniques.	Configure & operate the product.

**Testing notes**

**Tests**

**Problems Found**

Exploratory Forks



New test ideas occur continually during an ET session.

Exploratory Testing

- You know all the *techniques* you need to do good exploratory testing.
- The challenge is to develop your *thinking* about testing, so that you apply useful techniques at appropriate times.

By our *thinking*, we can compensate for a difficult project environment.

Instead of this... consider this.

- | | |
|--------------------------|------------------------------|
| – complete specs | – implicit specs & inference |
| – quantifiable criteria | – meaningful criteria |
| – protected schedule | – risk-driven iterations |
| – early involvement | – good working relationship |
| – zero defect philosophy | – good enough quality |
| – complete test coverage | – enough information |

Heuristic thinking

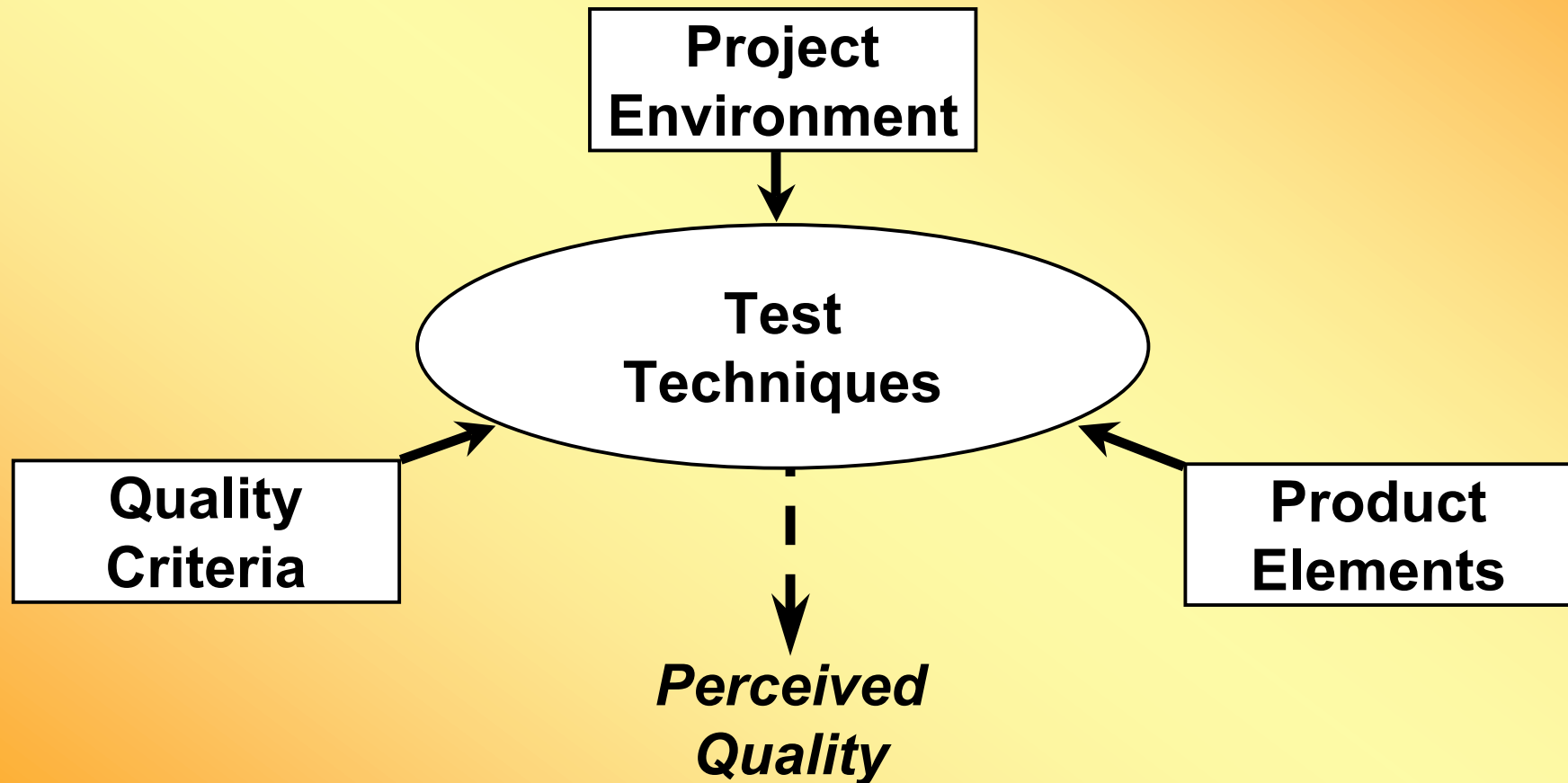
- A heuristic is a fallible idea or method that may help solve a problem.
- We've worked with several heuristics so far, including heuristics for:
 - Determining whether the program passed or failed a test (*all oracles are heuristics*).
 - Explaining why a behavior appears wrong (*the consistency heuristics*)
 - Organizing thinking about a project as a whole (*the heuristic test strategy model, for organizing risk-based testing and specification analysis, and soon, for test selection*)
 - Extending domain-testing thinking to non-ordered domains (*the printer testing example illustrates this approach*)
- The attacks are all examples of a heuristic approach to testing. They are educated guesses about what would be useful (given a fallible theory of error, here is a type of test that will probably expose the anticipated problem.)

A fundamental heuristic:

Testing is the process of asking questions of the program

- Product
 - What is this product?
 - What can I control and observe?
 - What should I test?
- Tests
 - What would constitute a diversified and practical test strategy?
 - How can I improve my understanding of how well or poorly this product works?
 - If there were an important problem here, how would I uncover it?
 - What document to load? Which button to push? What number to enter?
 - How powerful is this test?
 - What have I learned from this test that helps me perform powerful new tests?
 - What just happened? How do I examine that more closely?
- Problems
 - What quality criteria matter?
 - What kinds of problems might I find in this product?
 - Is what I see, here, a problem? If so, why?
 - How important is this problem? Why should it be fixed?

Heuristic Model for Test Design



Heuristic Model for Test Design

- We've used this model to organize failure-modes and to analyze specifications.
- We can **also** use it to guide test design overall.
 - If you're out of ideas about what to test,
 - Pick the focus of your tests by sampling your *Product's Elements* and considering which ones (or which interesting combinations) haven't yet been sufficiently tested, OR
 - Pick a theme for a set of tests by reviewing the *Quality Attributes* and considering which ones are important for the project that haven't yet been thoroughly tested. (Then pick the product elements that are interesting to test for this attribute, such as testing the product Preferences dialog (an element) for accessibility (a quality attribute).
 - If you're not sure what to test *for*
 - Review a failure mode catalog for examples of ways your product might fail.
 - When considering *how to test*,
 - Look carefully at your *Project Factors*, for suggestions of factors that might make one approach easier or less expensive or in some other way more appropriate than another.

The Plunge in and Quit Heuristic

Whenever you are called upon to test something very complex or frightening, **plunge in!**
After a little while, if you are very confused or find yourself stuck, **quit!**

- This is a tool for overcoming fear of complexity.
- Often, a testing problem isn't as hard as it looks.
- Sometimes it *is* as hard as it looks, and you need to quit for a while and consider how to tackle the problem.
- It may take several plunge & quit cycles to do the job.

Exploration Trigger Heuristic: No Questions

If you find yourself without any questions, ask yourself “Why don’t I have any questions?”

If you don’t have any *issues or concerns* about product testability or test environment, that itself may be a critical issue.

When you are uncautious, uncritical, or uncurious, that’s when you are most likely to let important problems slip by. *Don’t fall asleep!*

Observation vs. Inference

- Observation and inference are easily confused.
- Observation is direct sensory data, but on a very low level it is guided and manipulated by inferences and heuristics.
- You sense very little of what there is to sense.
- You remember little of what you actually sense.
- Some things you think you see in one instance may be confused with memories of other things you saw at other times.
- **It's easy to miss bugs that occur right in front of your eyes.**
- **It's easy to think you "saw" a thing when in fact you merely inferred that you must have seen it.**

What can you do about it?

Observation vs. Inference

Here's what:

- Accept that we're all fallible, but that we can learn to be better observers by learning from mistakes.
- Pay special attention to incidents where someone notices something you could have noticed, but did not.
- Don't strongly commit to a belief about any important evidence you've seen only once.
- Whenever you describe what you experienced, notice where you're saying what you saw and heard, and where you are instead jumping to a conclusion about "what was really going on."
- Where feasible, look at things in more than one way, and collect more than one kind of information about what happened (such as repeated testing, paired testing, loggers and log files, or video cameras).

ET Done Well is a Structured Process

- Exploratory testing, as we teach it, is a structured process conducted by a skilled tester, or by less skilled testers or users working under reasonable supervision.
- The structure of ET comes from:
 - Test design heuristics
 - Chartering
 - Time boxing
 - Perceived product risks
 - The nature of specific tests
 - The structure of the product being tested
 - The process of learning the product
 - Development activities
 - Constraints and resources afforded by the project
 - The skills, talents, and interests of the tester
 - The overall mission of testing

**In other words,
it's not “random”,
but reasoned.**

ET is an Adaptive Process

- Exploratory testing decentralizes the testing problem.
- Instead of trying to solve it:
 - only before test execution begins.
 - by investing in expensive test documentation that tends to reduce the total number of tests that can be created.
 - only via a designer who is not necessarily the tester.
 - while trying to eliminate the variations among testers.
 - completely, and all at once.
- It is solved:
 - over the course of the project.
 - by minimizing the need for expensive test documentation so that more tests and more complex tests can be created with the same effort.
 - via testers who may also be test designers.
 - by taking maximum advantage of variations among testers.
 - incrementally and cyclically.

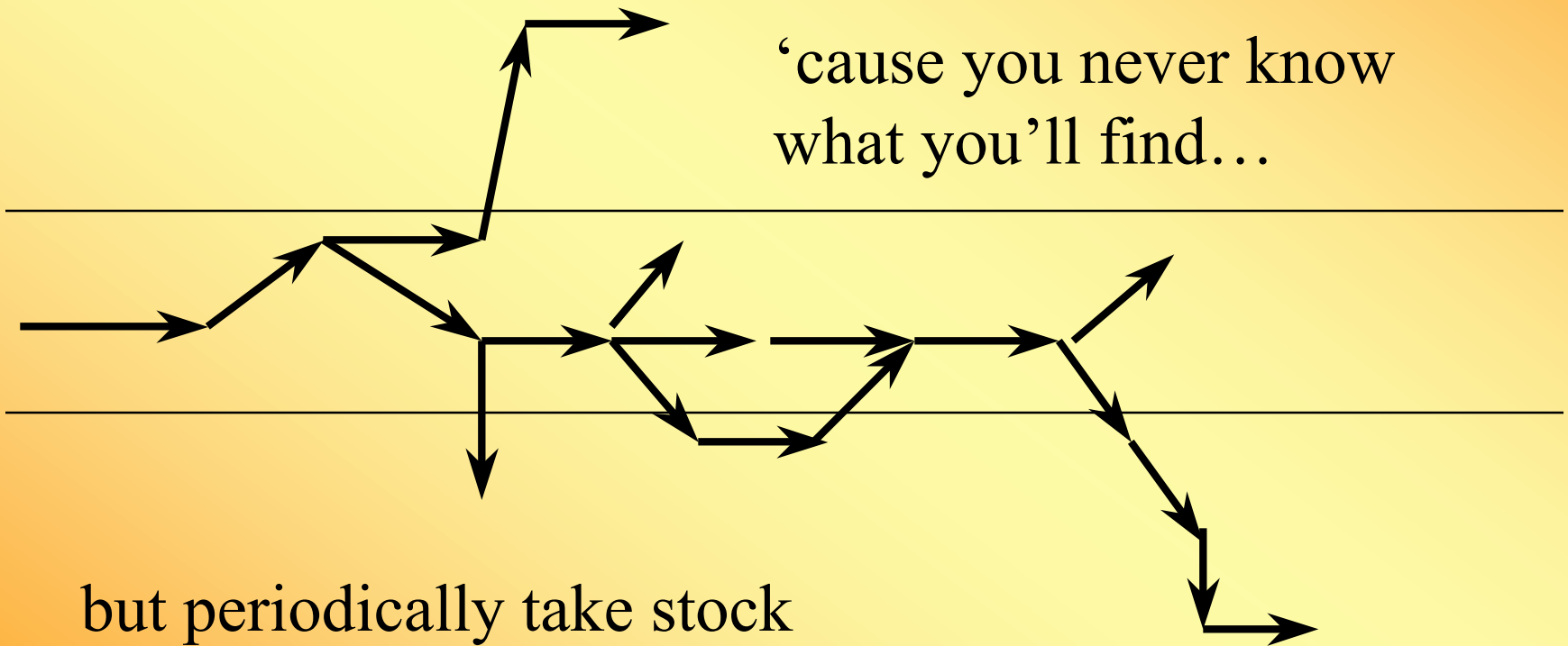
Thinking About Coverage

- Testers with less expertise...
 - Think about coverage mostly in terms of what they can see.
 - Cover the product indiscriminately.
 - Avoid questions about the completeness of their testing.
 - Can't reason about how much testing is enough.
- Better testers are more likely to...
 - Think about coverage in many dimensions.
 - Maximize diversity of tests while focusing on areas of risk.
 - Invite questions about the completeness of their testing.
 - Lead discussions on what testing is needed.

Lateral Thinking

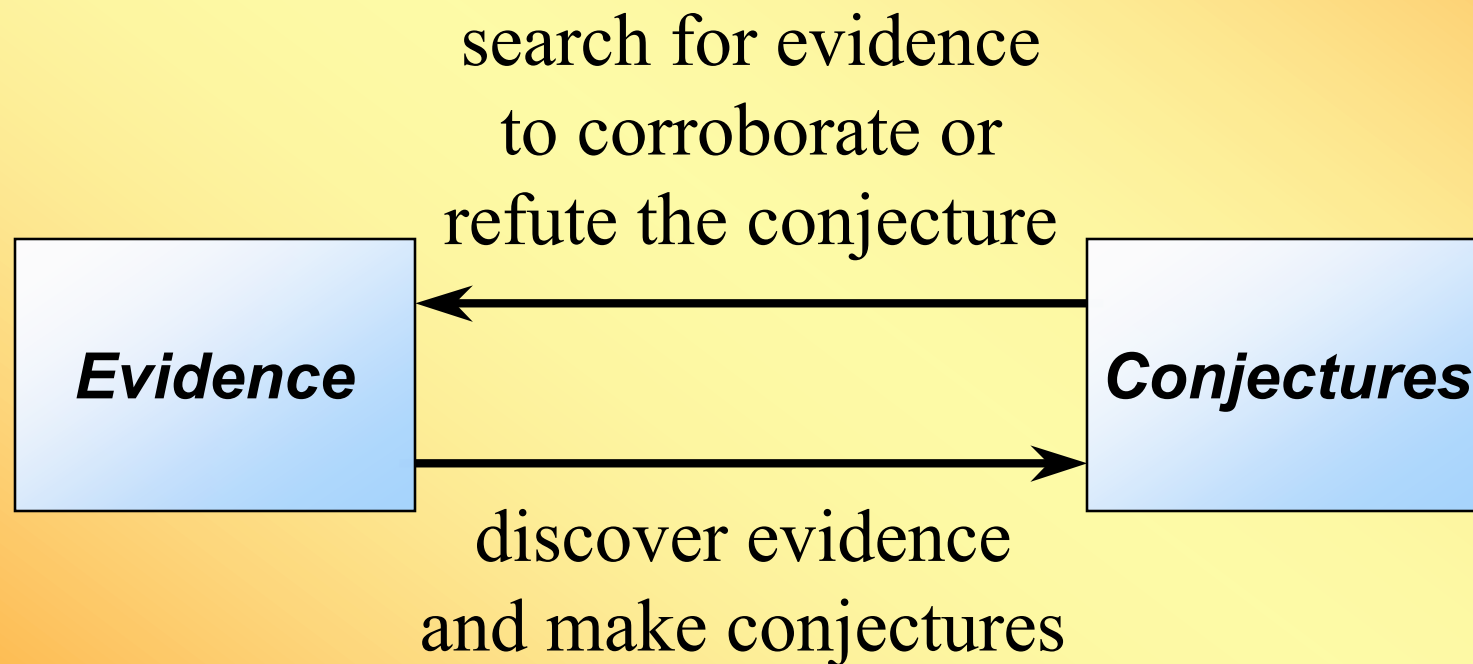
Let yourself be distracted...

‘cause you never know
what you’ll find...



but periodically take stock
of your status against your mission

Forward-Backward Thinking



Common Concerns About ET

- **Concerns**

- We have a long-life product and many versions, and we want a good corporate memory of key tests and techniques. Corporate memory is at risk because of the lack of documentation.
- The regulators would excommunicate us. The lawyers would massacre us. The auditors would reject us.
- We have specific tests that should be rerun regularly.

- **Replies**

- So, use a balanced approach, not purely exploratory.
- Even if you script all tests, you needn't outlaw exploratory behavior.
- Let no regulation or formalism be an excuse for bad testing.

Common Concerns About ET

- **Concerns**

- Some tests are too complex to be kept in the tester's head. The tester has to write stuff down or he will not do a thorough or deep job.

- **Replies**

- There is no inherent conflict between ET and documentation.
- There *is* often a conflict between writing *high quality* documentation and doing ET when both must be done at the *same moment*. But why do that?
- Automatic logging tools can solve part of the problem.
- Exploratory testing can be aided by any manner of test tool, document, or checklist. It can even be done from detailed test scripts.

Common Concerns About ET

- **Concerns**

- ET works well for expert testers, but we don't have any.

- **Replies**

- Detailed test procedures do not solve that problem, they merely obscure it, like perfume on a rotten egg.
- Our goal as test managers should be to develop skilled testers so that this problem disappears, over time.
- Since ET requires test design skill in some measure, ET management must constrain the testing problem to fit the level and type of test design skill possessed by the tester.
- We constrain the testing problem by personally supervising the testers, and making use of concise documentation, NOT by using detailed test scripts. *Humans make poor robots.*

Common Concerns About ET

- **Concerns**

- How do I tell the difference between bluffing and exploratory testing?
- If I send a scout and he comes back without finding anything, how do I know he didn't just go to sleep behind some tree?

- **Replies**

- You never know for sure— just as you don't know if a tester truly followed a test procedure.
- It's about reputation and relationships. Managing testers is like managing executives, not factory workers.
- Give novice testers short leashes; better testers long leashes. An expert tester may not need a leash at all.
- Work *closely* with your testers, and these problems go away.

Better Thinking

- *Conjecture and Refutation*: reasoning without certainty.
- *Abductive Inference*: finding the best explanation among alternatives.
- *Lateral Thinking*: the art of being distractible.
- *Forward-backward thinking*: connecting your observations to your imagination.
- *Heuristics*: applying helpful problem-solving short cuts.
- *De-biasing*: managing unhelpful short cuts.
- *Pairing*: two testers, one computer.
- *Study other fields*. Example: Information Theory.

Black Box Software Testing

Exploratory Testing in Pairs

Paired Exploratory Testing--Acknowledgment

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

We 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.

Black Box Software Testing

Regression Testing

Assigned Reading:

- Marick, *How Many Bugs Do Regression Tests Find?*
- Beck, *Test-Driven Development By Example*

Suggested Readings

- Hendrickson, *Difference Between Test Automation Failure & Success*
- Kaner, *Avoiding Shelfware: A Manager's View of GUI Test Automation*
- Pettichord, *Success with Test Automation*

Regression testing

- Tag line: “*Repeat testing after changes.*”
 - But scratch the surface and you find three fundamentally different visions:
 - *Procedural*: Run the same tests again
 - *Risk-oriented*: Expose errors caused by change
 - *Refactoring support*: Help the programmer discover implications of her code changes.
- Fundamental question or goal
 - Good regression testing gives clients confidence that they can change the product (or product environment).

Risk-oriented regression testing

- Tag line
 - “*Test 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.)

Risk-oriented regression testing

In this approach, we might re-use old tests or create new ones. Often, we retest an area or function with tests of increasing power (perhaps by combining them with other functions). The focus of the technique is on testing for side effects of change, not the inventory of old tests. Here are examples of a few common ways to test a program more harshly while retesting in the same area.

- Do more iterations (one test hits the same function many times).
- Do more combinations (interactions among variables, including the function under test's variables).
- Do more things (sequences of functions that include the function under test).
- Methodically cover the code (all N-length sequences that include the function under test; all N-wide combinations that include the function under test's variables and variables that are expected to interact with or constrain or be constrained by the function under test).
- Look for specific errors (such as similar products' problems) that involve the function under test.
- Try other types of tests, such as scenarios, that involve the function under test.
- Try to break it (take a perverse view, get creative).

» Thanks to Doug Hoffman for a draft of this list

Refactoring support: Change detectors

- Tag line
 - *"Change detectors"*
- Fundamental question or goal
 - Support refactoring: Help the programmer discover implications of her code changes.
- Paradigmatic case(s)
 - *Test-driven development using glass-box testing tools like junit, httpunit, and fit.*

NOTES:

- The programmer creates these tests and runs them every time she compiles the code.
- If a test suite takes more than 2 minutes, the programmer might split tests into 2 groups (tests run at every compile and tests run every few hours or overnight).
- The intent of the tests is to exercise every function in interesting ways, so that when the programmer refactors code, she can quickly see
 - (a) what would break if she made a change to a given variable, data item or function or
 - (b) what she did break by making the change.

Refactoring support: Change detectors

This is out of scope for Testing 1, but a substantial part of the Testing 2 course.

There are enormous practical differences between system-level black-box regression testing and unit-level (or integration-level) glass-box regression testing

- In the unit test situation, the programmer (not an independent tester) writes the tests, typically before she writes the code. The testing focuses the programming, yielding better code in the first place.
- In the unit test case, when the programmer makes a change that has a side-effect, she immediately discovers the break and fixes it. There is no communication cost. You don't have (as you would in black box testing) a tester who discovers a bug, replicates it, reports it, and then a project manager who reads the report, maybe a triage team who study the bug and agree it should be fixed, a programmer who has to read the report, troubleshoot the problem, fix it, file the fix, a tester who has to retest the bug to determine that the fix really fixed the problem and then close the bug. All labor-hours considered, this can easily cost 4 hours of processing time, compared to a few minutes to fix a bug discovered at the unit level.
- In the black box case, test case maintenance costs are high, partially because the same broken area of code might be involved in dozens of system level tests. And partially because it takes a while for the black box tester to understand the implications of the code changes (which he doesn't see). The programmer fixes tests that are directly tied to the changes she makes, and she sees the tests break as soon as she makes the change, which helps her reappraise whether the change she is making is reasonable.
- Beck, *Test-Driven Development By Example*
- Astels, *Test-Driven Development*
- Link, *Unit Testing in Java*
- Fowler, *Refactoring*

Procedural regression testing

- Tag line
 - *"Run the same tests again"*
- Paradigmatic cases
 - Manual, scripted regression testing
 - Automated GUI regression testing
 - Smoke testing (manual or automated)

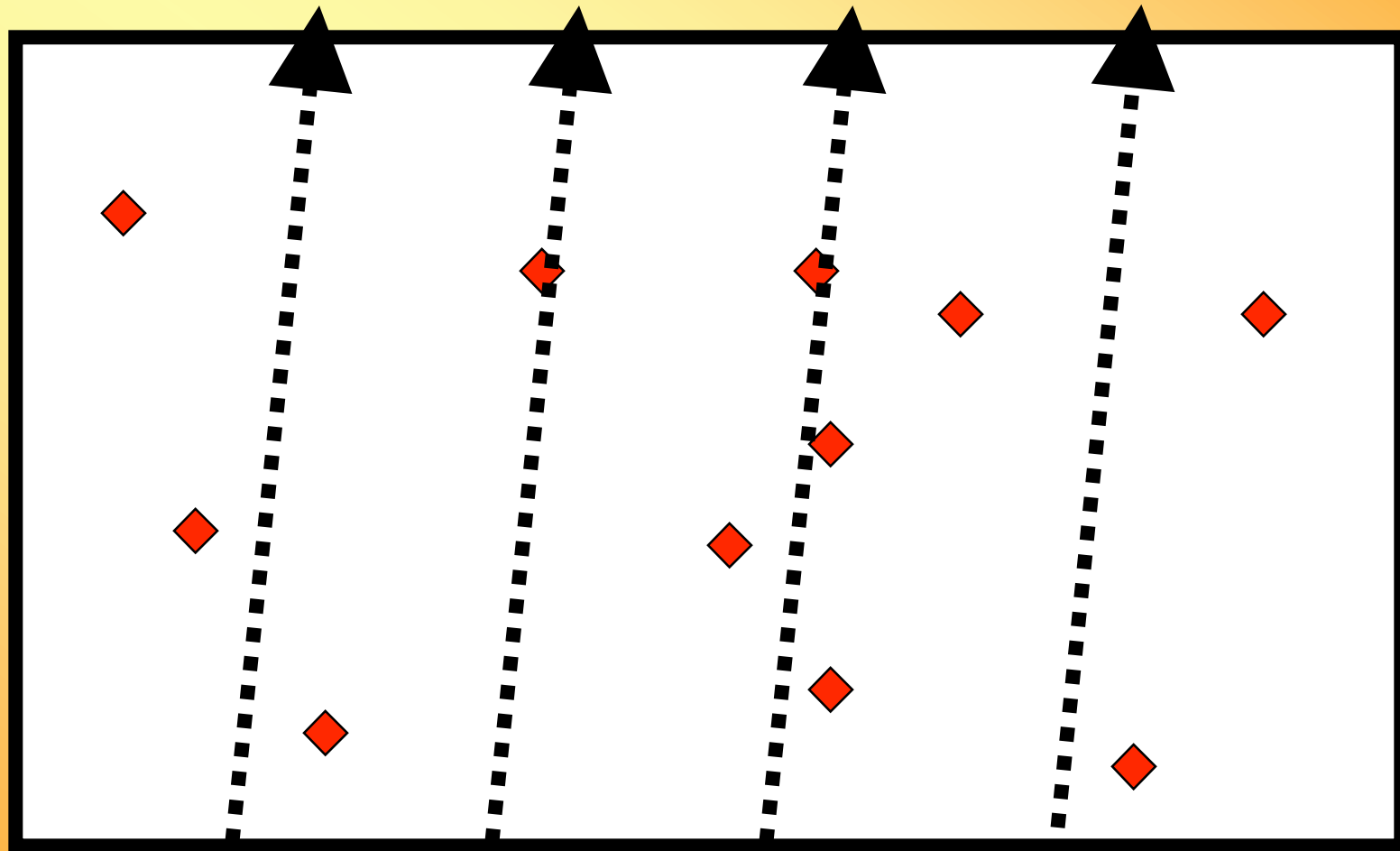
Procedural regression testing

- Benefits
 - The tests exist already (no need for new design, or new implementation, but there will be maintenance cost)
 - Many regulators and process inspectors like them
 - Because we are investing in re-use, we can afford to take the time to craft each test carefully, making it more likely to be powerful in future use
 - This is the dominant paradigm for automated testing, so it is relatively easy to justify and there are lots of commercial tools
 - Implementation of automated tests is often relatively quick and easy (though maintenance can be a nightmare)
 - *We'll look more closely at implementation details in the discussion of automation. For now, let's look at the conceptual strengths and weaknesses of this approach.*

Automating GUI regression testing

- This is the most commonly discussed automation approach:
 1. create a test case
 2. run it and inspect the output
 3. if the program fails, report a bug and try again later
 4. if the program passes the test, save the resulting outputs
 5. 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.

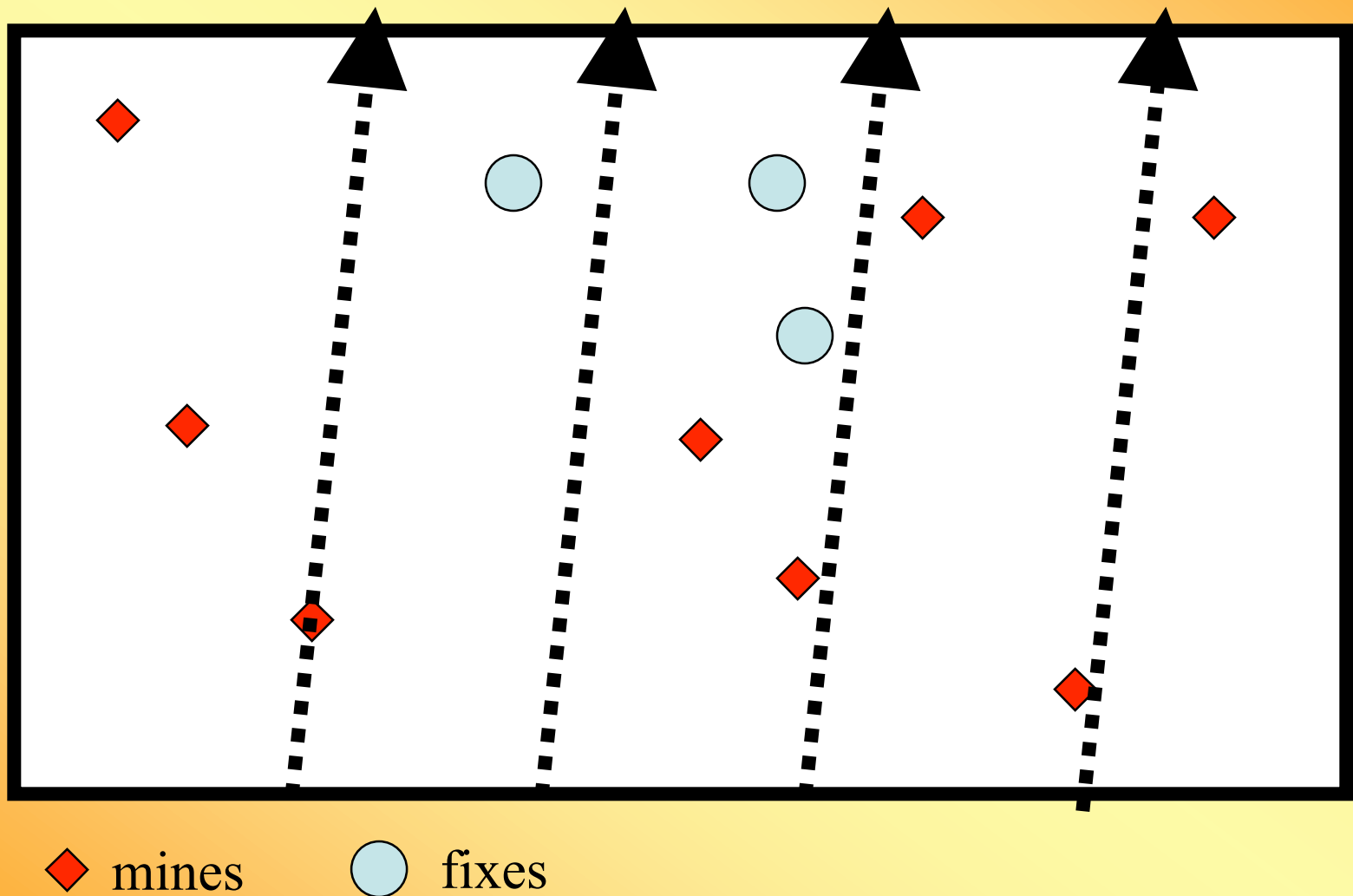
An analogy: Clearing mines



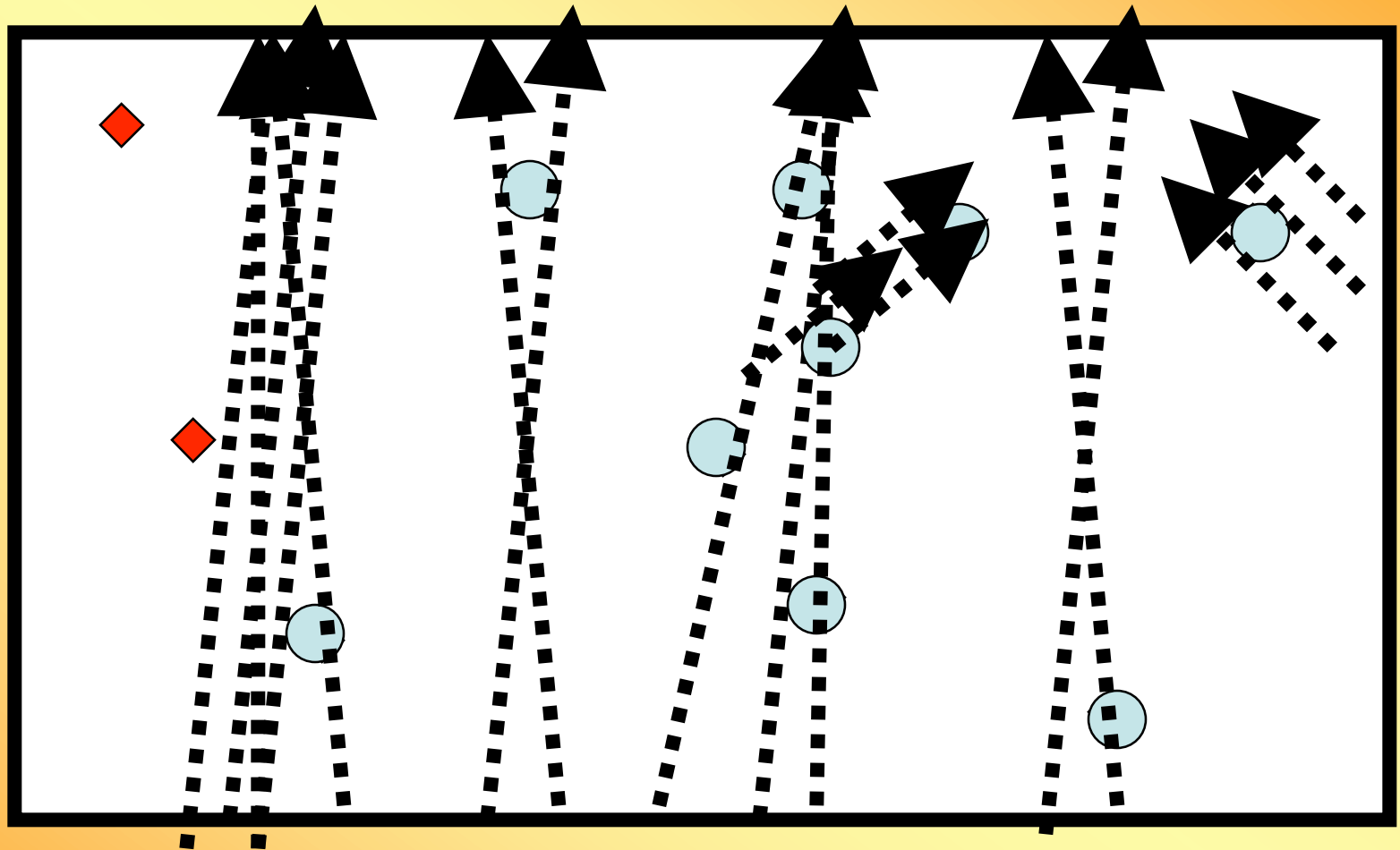
◆ mines

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



◆ mines

● fixes

Procedural regression testing

- 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.

Automated GUI regression testing

- Look back at the minefield analogy
- Are you convinced that variable tests will find more bugs under all circumstances?
 - *If so, why would people do repeated tests?*

Generate 10 counter-examples to the minefield analogy.



The Regression Testing paradigm

- 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.

Is this really automation?

- | | | |
|---------------------|----|--|
| • Analyze product | -- | human |
| • Design test | -- | human |
| • Run test 1st time | -- | human |
| • Evaluate results | -- | human |
| • Report 1st bug | -- | human |
| • Save code | -- | human |
| • Save result | -- | human |
| • Document test | -- | human |
| • Re-run the test | -- | MACHINE |
| • Evaluate result | -- | machine plus
human if there's
any mismatch |
| • Maintain result | -- | human |

**Woo-hoo! We
really get the
machine to do a
whole lot of our
work!**

(Maybe, but not
this way.)

GUI automation is expensive

- Test case creation is expensive. Estimates run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (LAWST).
- You usually have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation
- Automation can delay testing, adding even more cost (albeit hidden cost.)
- Excessive reliance leads to the 20 questions problem. (Fully defining a test suite in advance, before you know the program's weaknesses, is like playing 20 questions where you have to ask all the questions before you get your first answer.)

GUI automation pays off late

- Regression testing has low power because:
 - Rerunning old tests that the program has passed is less powerful than running new tests.
- Maintainability is a core issue because our main payback is usually in the next release, not this one.

Test automation is programming

- Win NT 4 had 6 million lines of code, and 12 million lines of test code
- Common (and often vendor-recommended) design and programming practices for automated testing are appalling:
 - **Embedded constants**
 - No modularity
 - ***No source control***
 - No documentation
 - No requirements analysis
 - No wonder we fail.

Common mistakes in GUI automation

- Don't underestimate the cost of automation.
- Don't underestimate the need for staff training.
- Don't expect to be more productive over the short term.
- Don't spend so much time and effort on regression testing.
- Don't use instability of the code as an excuse.
- Don't put off finding bugs in order to write test cases.
- Don't write simplistic test cases.
- Don't shoot for "100% automation."
- Don't use capture/replay to create tests.
- Don't write isolated scripts in your spare time.

Common mistakes in GUI automation

- Don't create test scripts that won't be easy to maintain over the long term.
- Don't make the code machine-specific.
- Don't fail to treat this as a genuine programming project.
- Don't "forget" to document your work.
- Don't deal unthinkingly with ancestral code.
- Don't give the high-skill work to outsiders.
- Don't insist that all of your testers be programmers.
- Don't put up with bugs and crappy support for the test tool.
- Don't forget to clear up the fantasies that have been spoonfed to your management.

Requirements analysis

Automation requirements are not just about the software under test and its risks. To understand what we're up to, we have to understand:

- Software under test and its risks
- The development strategy and timeframe for the software under test
- How people will use the software
- What environments the software runs under and their associated risks
- What tools are available in this environment and their capabilities
- The regulatory / required recordkeeping environment
- The attitudes and interests of test group management.
- The overall organizational situation

Requirements analysis

- Requirement: “Anything that drives design choices.”
- The paper (Avoiding Shelfware) lists 27 questions. For example,

Will the user interface of the application be stable or not?

- Let’s analyze this. The reality is that, in many companies, the UI changes late.
- Suppose we’re in an extreme case. Does that mean we cannot automate cost effectively? No. It means that we should do only those types of automation that will yield a fast return on investment.

Requirements questions

- Will the user interface of the application be stable or not?
- To what extent are oracles available?
- To what extent are you looking for delayed-fuse bugs (memory leaks, wild pointers, etc.)?
- Does your management expect to recover its investment in automation within a certain period of time? How long is that period and how easily can you influence these expectations?
- Are you testing your own company's code or the code of a client? Does the client want (is the client willing to pay for) reusable test cases or will it be satisfied with bug reports and status reports?
- Do you expect this product to sell through multiple versions?
- Do you anticipate that the product will be stable when released, or do you expect to have to test Release N.01, N.02, N.03 and other bug fix releases on an urgent basis after shipment?

Requirements questions

- Do you anticipate that the product will be translated to other languages? Will it be recompiled or relinked after translation (do you need to do a full test of the program after translation)? How many translations and localizations?
- Does your company make several products that can be tested in similar ways? Is there an opportunity for amortizing the cost of tool development across several projects?
- How varied are the configurations (combinations of operating system version, hardware, and drivers) in your market? (To what extent do you need to test compability with them?)
- What level of source control has been applied to the code under test? To what extent can old, defective code accidentally come back into a build?
- How frequently do you receive new builds of the software?
- Are new builds well tested (integration tests) by the developers before they get to the tester?

Requirements questions

- To what extent have the programming staff used custom controls?
- How likely is it that the next version of your testing tool will have changes in its command syntax and command set?
- What are the logging/reporting capabilities of your tool? Do you have to build these in?
- To what extent does the tool make it easy for you to recover from errors (in the product under test), prepare the product for further testing, and re-synchronize the product and the test (get them operating at the same state in the same program).
- (In general, what kind of functionality will you have to add to the tool to make it usable?)
- Is the quality of your product driven primarily by regulatory or liability considerations or by market forces (competition)?
- Is your company subject to a legal requirement that test cases be demonstrable?

Requirements questions

- Will you have to be able to trace test cases back to customer requirements and to show that each requirement has associated test cases?
- Is your company subject to audits or inspections by organizations that prefer to see extensive regression testing?
- If you are doing custom programming, is there a contract that specifies the acceptance tests? Can you automate these and use them as regression tests?
- What are the skills of your current staff?
- Do you have to make it possible for non-programmers to create automated test cases?
- To what extent are cooperative programmers available within the programming team to provide automation support such as event logs, more unique or informative error messages, and hooks for making function calls below the UI level?
- What kinds of tests are really *hard* in your application? How would automation make these tests easier to conduct?

You *Can* Plan for Short Term ROI

- Smoke testing
- Configuration testing
- Variations on a theme
- Stress testing
- Load testing
- Life testing
- Performance benchmarking
- Other tests that extend your reach

Four sometimes-successful GUI regression architectures

- Quick & dirty
- Equivalence testing
- Framework
- Data-driven

Equivalence testing

- A/B comparison
- Random tests using an oracle
- Regression testing is the weakest form

Framework-based architecture

Frameworks are code libraries that separate routine calls from designed tests.

- modularity
- reuse of components
- hide design evolution of UI or tool commands
- partial salvation from the custom control problem
- independence of application (the test case) from user interface details (execute using keyboard? Mouse? API?)
- important utilities, such as error recovery

For more on frameworks, see Linda Hayes' book on automated testing, Tom Arnold's book on Visual Test, and Mark Fewster & Dorothy Graham's book "Software Test Automation."

Data-driven tests

- Variables are data
- Commands are data
- UI is data
- Program's state is data
- Test tool's syntax is data

Data-driven architecture

- In test automation, there are three interesting programs:
 - The software under test (SUT)
 - The automation tool that executes the automated test code
 - The test code (test scripts) that define the individual tests
- From the point of view of the automation software,
 - The SUT's variables are data
 - The SUT's commands are data
 - The SUT's UI is data
 - The SUT's state is data
- Therefore it is entirely fair game to treat these implementation details of the SUT as values assigned to variables of the automation software.
- Additionally, we can think of the externally determined (e.g. determined by you) test inputs and expected test results as data.
- Additionally, if the automation tool's syntax is subject to change, we might rationally treat the command set as variable data as well.

Data-driven architecture

- In general, we can benefit from separating the treatment of one type of data from another with an eye to:
 - optimizing the maintainability of each
 - optimizing the understandability (to the test case creator or maintainer) of the link between the data and whatever inspired those choices of values of the data
 - minimizing churn that comes from changes in the UI, the underlying features, the test tool, or the overlying requirements

Data-driven architecture: Calendar example

Imagine testing a calendar-making program.

- The look of the calendar, the dates, etc., can all be thought of as being tied to physical examples in the world, rather than being tied to the program. If your collection of cool calendars wouldn't change with changes in the UI of the software under test, then the test data that define the calendar are of a different class from the test data that define the program's features.
 1. Define the calendars in a table. This table should not be invalidated across calendar program versions. Columns name features settings, each test case is on its own row.
 2. An interpreter associates the values in each column with a set of commands (a test script) that execute the value of the cell in a given column/row.
 3. The interpreter itself might use “wrapped” functions, i.e. make indirect calls to the automation tool's built-in features.

Data-driven architecture: Calendar example

- This is a good design from the point of view of optimizing for maintainability because it separates out four types of things that can vary independently:
 1. The descriptions of the calendars themselves come from real-world and can stay stable across program versions.
 2. The mapping of calendar element to UI feature will change frequently because the UI will change frequently. The mappings (one per UI element) are written as short, separate functions that can be maintained easily.
 3. The short scripts that map calendar elements to the program functions probably call sub-scripts (think of them as library functions) that wrap common program functions. Therefore a fundamental change in the program might lead to a modest change in the software under test.
 4. The short scripts that map calendar elements to the program functions probably also call sub-scripts (think of them as library functions) that wrap functions of the automation tool. If the tool syntax changes, maintenance involves changing the wrappers' definitions rather than the scripts.

Data driven architecture

- Note with this example:
 - we didn't run tests twice
 - we automated execution, not evaluation
 - we saved SOME time
 - we focused the tester on design and results, not execution.
- Other table-driven cases:
 - automated comparison can be done via a pointer in the table to the file
 - the underlying approach runs an interpreter against table entries.
 - Hans Buwalda and others use this to create a structure that is natural for non-tester subject matter experts to manipulate.
 - Ward Cunningham's FIT tool enables the same type of testing, but below the level of the UI. See <http://fit.c2.com/>

Think about:

- Automation is software development.
- Regression automation is expensive and can be inefficient.
- Automation need not be regression--you can run new tests instead of old ones.
- Maintainability is essential.
- Design to your requirements.
- Set management expectations with care.

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