Carts Before Horses: Using Preparatory Exercises to Motivate Lecture Material

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

Workshop on Teaching Software Testing

Melbourne, FL

January, 2004

FLORIDA INSTITUTE OF TECHNOLOGY ★ 1958 ★

CENTER FOR SOFTWARE TESTING EDUCATION AND RESEARCH

Rationale

- Lectures are effective for ***conveying*** a standard set of information in a time-controlled way. They also help the skilled teacher establish rapport and credibility with the class. The skilled teacher can also use them to convey attitudes and life lessons.
- Many of us have evolved into very effective, popular lecturers. One of the challenges that some of us face is retooling--shifting from relying on a lecture format that we're pretty good at, to something else.
- Unfortunately, lectures can be poor vehicles for communication (just because you speak it doesn't mean they understand it or learn the specific lessons you hope they get from it), or for developing the students' technical or communication skills.
- I am not ready to abandon lectures, but I am looking for techniques to increase student attention and interest in the things I present.
- One of my current tactics is to present problems to students before they have the specific materials they need to solve them. After they chew on the problems for a while, I use the lecture to either present a solution or (preferably) enable them to develop their own solution.

Rationale

- Computer Science students are generally willing to play along with puzzles, so even an arbitrary puzzle can orient students to a lecture in which we explain the solution.
- I prefer to use authentic exercises, based on real situations. They may not look authentic in the puzzle, but I take pains to tie them back to reality in the lecture. Over time, I believe this builds trust and credibility. (And lets me get away with more complex and perhaps more initially-odd-seeming puzzles.)

Simple Examples: Pre-tests

You are testing a program that includes an Integer Square Root function. It interprets the contents of a 32-bit word as an unsigned integer and then computes the square root of the integer, as a floating point number.

A What values can you input to this function?

B Can you imagine any invalid inputs to this function, inputs that should cause the function to yield an error message?

C If you were to test all the inputs to this function, how many tests would there be?

D How long do you think it would take you to run all these tests?

E How would you describe how to test this function? Describe your thinking about your possible testing strategies.

F Describe the tests that you would actually run and explain why these are useful or interesting tests.

G Would you add more tests if you wanted to test the program thoroughly and be confident that there are no bugs?

H How would you tell whether the program passed or failed your tests?

I Have you ever done this type of testing? If so, when? Please describe your experience.

J Have you ever used tools that would help you do this testing? What tools? What did you do with them?

Simple Examples: Pre-tests

Suppose you were writing a text editing program that would display fonts properly on the screen. How would you test whether the font display is correct?

Here is some terminology:

A ***typeface*** is a style or design of a complete set of printable characters. For example, these are all in the typeface "Times New Roman":

Times New Roman, size 10 points	Times New Roman, size 12 points
Times New Roman, size 14 points	Times New Roman, size 16 points

The following are in the Helvetica typeface:

Helvetica, size10 points	Helvetica, size 12 points
Helvetica, size 14 points	Helvetica, size 16 points

When you add a size to a typeface, you have a font. The line above this is printed in a Helvetica 12 point font.

A How could you determine whether the editor is displaying text in the correct typeface?

B How could you determine whether the editor is displaying text in the right size?

C How would you test whether the editor displays text correctly (in any font that the user specifies)? Describe your thinking about this problem and the strategies you are considering.

A More Complex Exercise: Open Office Test Docs.

Sun Microsystems has decided to provide financial support for development of a Release 2.0 version of Open Office.

Along with cash, they will supply staff (as they have done for some previous releases). They want to understand the commitment they should make to doing or managing the testing effort, and they have decided to retain a consulting firm to help them evaluate the effort and perhaps to help them set up the testing work or even supervise it on a long-term basis.

To select a consultant, they are running two competitions. The first competition involves many consulting firms, who are each given the same short assignment. The best four of those will be invited to California for a longer term assignment and the best one of those will get the contract.

We are in the first competition. You are a member of one of three teams being tested today. (Many other teams will be tested elsewhere, on other days.) The odds are that only one of these teams will go to California.

A More Complex Exercise: Open Office Test Docs.

Your task is to advise Sun on the type(s) and depth of test documentation needed for this project. What do they need and why do they need it?

Three executives are here from Sun today. One will go with each group, to that group's meeting room. You can ask Sun executives any questions, (they might not be able to answer some questions, but you can ask).

- After about 20 minutes, the executive will leave your room.
- After 35 minutes, you will come back to the main meeting room and each group will present its answer to the Sun executives.
- You will have 10 minutes. You can spend your 10 minutes making your presentation, answering questions, or in closing remarks after the other two groups have made their main presentation. You have 10 minutes, but only 10 minutes.

A More Complex Exercise: Open Office Test Docs.

Group Presentations

A More Complex Exercise: Open Office Test Docs.

- After the group presentations (and critiques), I present my notes on requirements analysis for test documentation. This takes 30-45 minutes.
- The groups then meet again for 35 minutes, prepare their presentations, and then present and debrief.

A More Complex Exercise: Open Office Test Docs.

Benefits:

- Lessons learned:
  - Test documentation is a product, subject to context-specific needs, wishes, and constraints.
  - Different stakeholders have different needs, wishes and constraints and often want incompatible solutions.
  - If you don't check in with your stakeholders, you're probably going to deliver the wrong thing(s).
  - There are lots of types and levels of depth / focus / scope / objectives of test documentation.
- Skills practiced
  - Oral presentation
  - Group planning
  - Requirements questioning

A More Complex Exercise: Exploration & Bug Reporting

PREFACE

- This exercise has worked well for a commercial class, whose students had used plenty of black box test techniques on the job.
- Paired exploratory testing has worked well for mixed groups (experienced paired with novice). We suspect that novice/novice pairs need introductory training in test techniques, to give them a sense of the scope open to them.
- In class, we used Open Office. Here, we're illustrating it with MS Word (because you probably know that better).
- It's important to pick a feature that students have a chance of understanding quickly. If the students are inexperienced, pick a target-rich feature, like outlining (including bullets, numbering, headings, and headings within tables).
- It's very important to assign students numbers after they split into groups. You will resplit them twice more and need to avoid prior exposures.

A More Complex Exercise: Exploration & Bug Reporting

- We're going to test the Microsoft Word's outlining feature (including bullets, numbering, headings, and headings within tables).
- To start this, we will split into pairs. Each pair will test together. Take careful notes -- in the next part of this exercise, you will enter your bug reports into the database.

A More Complex Exercise: Exploration & Bug Reporting

- **To do paired exploratory testing, start with a charter**
  - A typical exploratory testing session lasts about 60-90 minutes. Ours will run 50 minutes.
  - The charter for a session might include what to test, what risks are involved, what tools to use, what testing tactics to use, what bugs to look for, what documents to examine, what outputs are desired, etc.
    - Some testers work from a detailed project outline, picking a task that will take a day or less.
    - Others create a flipchart page that outlines this session's work or the work for the next few sessions.

A More Complex Exercise: Exploration & Bug Reporting

- A quick summary of commonly used test techniques:
  - Risk-based testing
    - Failure modes and effects testing
    - Exploratory attacks
  - Specification-based testing
  - Domain testing
  - Function testing
  - Scenario testing
  - User
  - State model
  - Regression
  - High volume automation

A More Complex Exercise: Exploration & Bug Reporting

TESTING

TESTING

TESTING

A More Complex Exercise: Exploration & Bug Reporting

NOW LET'S REPORT THE BUGS

1. Find a new partner
2. Each of you should pick your two favorite bugs (from the ones you found)
3. For each bug, write a report that has two parts
   - Summary: a brief (one-line) description of the problem
   - Problem & How to Reproduce It
     - Describe the problem in a persuasive way (your report will, or will not, convince someone to spend time fixing this bug).
     - Describe the problem clearly--step by step, numbered steps, describing exactly how to reproduce the bug
4. Review your partner's bug reports and suggest improvements; fix your reports

A More Complex Exercise: Exploration & Bug Reporting

EVALUATING THE BUGS

You're now going to find yet another new partner and evaluate some bug reports. Before we do the evaluation, here are some guidelines for figuring out what is a good bug report.

- Bug advocacy
  - making bugs more compelling
    - clear statement of worst consequence
    - follow-up tests if useful
    - scope identified and clear
  - eliminating objections
    - reproducibility
    - lack of clarity (or just plain unintelligible)
    - insult
- Bug evaluation (see slides)

Instructor notes

First pairs

ab	cd	ef	gh

Second pairs

ac	bd	eg	fh

Third pairs

ae(gi)	bf(hj)	cg(km)	dh(ln)

- Pair ae(gi): Person (a) and Person (e) jointly review the bug reports submitted by person (g) and person (i).
- Note that (a) and (e) have not worked together and that neither (a) nor (e) has previously seen bugs from (g) or (i).

A More Complex Exercise: Exploration & Bug Reporting

EVALUATE BUGS

1. Pick a new partner (see instructor notes). You will role play.
   - One of you is the test manager. You want bugs reported competently, and you want bugs fixed.
   - One of you is the project manager. You expect bug reports to assist, and never interfere, with your staff's efficience in evaluating and fixing bugs. You want to ship the product on time, with the right features, within budget, and in all important respects, working.
2. Use the attached notes on evaluating bugs as a guide, a set of (all of them optional) ideas for evaluating the quality of the bug report.
3. For each bug, evaluate the quality of the bug report
4. Evaluate the significance of the bug
5. ON COMPLETION OF THE TASK, THE GROUP DEBRIEFS, DISCUSSING THE BUG REPORTS AND THE DYNAMICS OF REVIEWING THEM.

A More Complex Exercise: Exploration & Bug Reporting

- Challenges:
  - this is a long series of tasks
  - this requires prep lectures for students who have no background
  - this requires lab space for paired work
- What is learned:
  - role playing
  - communication skills
  - bug analysis skills
  - bug reporting skills
  - timeboxed exploratory testing
  - paired testing
  - chartering

The examples again,
With supporting lecture notes

Simple Examples: Pre-tests

You are testing a program that includes an Integer Square Root function. It interprets the contents of a 32-bit word as an unsigned integer and then computes the square root of the integer, as a floating point number.

A What values can you input to this function?

B Can you imagine any invalid inputs to this function, inputs that should cause the function to yield an error message?

C If you were to test all the inputs to this function, how many tests would there be?

D How long do you think it would take you to run all these tests?

E How would you describe how to test this function? Describe your thinking about your possible testing strategies.

F Describe the tests that you would actually run and explain why these are useful or interesting tests.

G Would you add more tests if you wanted to test the program thoroughly and be confident that there are no bugs?

H How would you tell whether the program passed or failed your tests?

I Have you ever done this type of testing? If so, when? Please describe your experience.

J Have you ever used tools that would help you do this testing? What tools? What did you do with them?

Black Box Software Testing

2004 Academic Edition

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.

Black Box Software Testing

Part 2
Complete Testing Is Impossible

the space to search is vast...
...and your resources are meager.

Complete testing?

- What ***do*** we mean by "complete testing"?
  - Complete "coverage": Tested every line / branch / basis path?
  - Testers not finding new bugs?
  - Test plan complete?

Complete testing ***must*** mean that, at the end of testing, you know there are no remaining unknown bugs.

  - After all, if there are more bugs, you can find them if you do more testing. So testing couldn't yet be "complete."

Let's consider the nature of the infinite set of tests

- There are enormous numbers of possible tests. To test everything, you would have to:
  - Test every possible input to every variable (including output variables and intermediate results variables).
  - Test every possible combination of inputs to every combination of variables.
  - Test every possible sequence through the program.
  - Test every hardware / software configuration, including configurations of servers not under your control.
  - Test every way in which any user might try to use the program.

» Read Testing Computer Software, p. 17 - 22

Inputs to single variables

Valid inputs

- –Doug Hoffman worked for MASPAR (the Massively Parallel computer, 64K parallel processors). This machine is used for mission-critical and life-critical applications.
- –To test the 32-bit integer square root function, Hoffman checked all values (all 4,294,967,296 of them). This took the computer about 6 minutes to run the tests and compare the results to an oracle.
- –There were 2 (two) errors, neither of them near any boundary. (The underlying error was that a bit was sometimes mis-set, but in most error cases, there was no effect on the final calculated result.) Without an exhaustive test, these errors probably wouldn't have shown up.
- –***What about the 64-bit integer square root?*** How could we find the time to run all of these? If we don't run them all, don't we risk missing some bugs?

Inputs to single variables

- Easter Eggs
  - Bizarre inputs, by design
- Edited inputs
  - These can be quite complex. How much editing is enough?
- Variations on input timing
  - Try entering the data very quickly, or very slowly. Enter them before, after, and during the processing of some other event, or just as the time-out interval for this data item is about to expire.
- Now, what about all the error handling that you can trigger with "invalid" inputs?
  - Think about Whittaker & Jorgensen's constraint-focused attacks (Whittaker, *How Software Fails)*
  - Think about Jorgensen's hostile data stream attacks

When people challenge extreme value tests…

“No user would do that.”

really means…

“No user I can *think of*, who I *like*, would do that *on purpose*.”

Who aren’t you thinking of?
Who don’t you like who might really use this product?
What might good users do by accident?

INFO TO GO

- Failures can lurk in incredibly obscure places.
- You could need 4 billion computations to find 2 errors caused by 1 bug.

Exhausting Your Test Options

by Douglas Hoffman

SOME TIME AGO, I WORKED AS QA MANAGER FOR a startup that was making massively parallel systems. Each computer system had between 1,024 and 65,536 processors. The most powerful systems we made were about twenty times as powerful as the largest single-CPU systems at the time (in raw CPU cycles). With that much raw CPU power, we could address some problems that were otherwise unthinkable at the time. (For example, one of our systems was used to correct some of the images from the Hubble Space Telescope while the lenses were being fixed.) That much raw CPU power also created some testing problems and opportunities that were unthinkable at the time. Let me give you a bit of background so you can better understand the challenges we faced.

The system processor array was laid out much like a spreadsheet, with rows and columns of processors that had neighbor processors at each of eight relative positions, as shown in Figure 1. The processors could send and receive data with each of the neighbors directly and with any other processor in the array indirectly. Each processor received the same machine instructions from the program, but had unique data. (Such machines are called "SIMD," for Single Instruction, Multiple Data.) SIMD machines work well for problems, such as weather modeling, that can be broken up into a large number of chunks of data, each of which is handled the same way. The algorithm is the same everywhere, but each processor's outcome is uniquely defined in its own private data.

We had several levels of software to deal with. There were some ordinary applications. There were programmer tools like compilers, debuggers, and data visualizers. There was an operating system that controlled the array. And there was even software embedded in the processor chips. Each instruction a processor executed was actually a small program, written in what's called "microcode." With all these levels to test, this was definitely an environment that kept the test and QA group hopping.

Dinner Napkins and Microcode

I was having dinner at the office one evening with one of the two principal architects of the system. (As is typical at most startups, we had dinner brought in five or six nights a week to encourage us to stay, reduce the time we had to take to eat, and provide more opportunities for us to talk shop.) We were in the final stages of defining the processor instruction set and were trading ideas about validating it. Our conversation eventually came around to the problem of dealing with the most complex machine instruction—square root.

Most of the machine instructions were very simple programs with a few dozen microcode steps. We had confidence in these from code inspections and use. The square root functions, however, were much more complex, running hundreds of steps. Inspection was complicated by the fact that only the two principal architects understood the details of the processor architecture and machine micro instruction set, so they were the only people qualified to analyze the code in depth. Others of us participated in some of the inspections, but were limited by our fuzzy understanding of the internals of the machines.

Between bites of dinner, we discussed the number of hours it was going to take to inspect the code and when we might be able to squeeze it into our schedules. As we bemoaned the loss of time and all the other work we had to do, an application program-

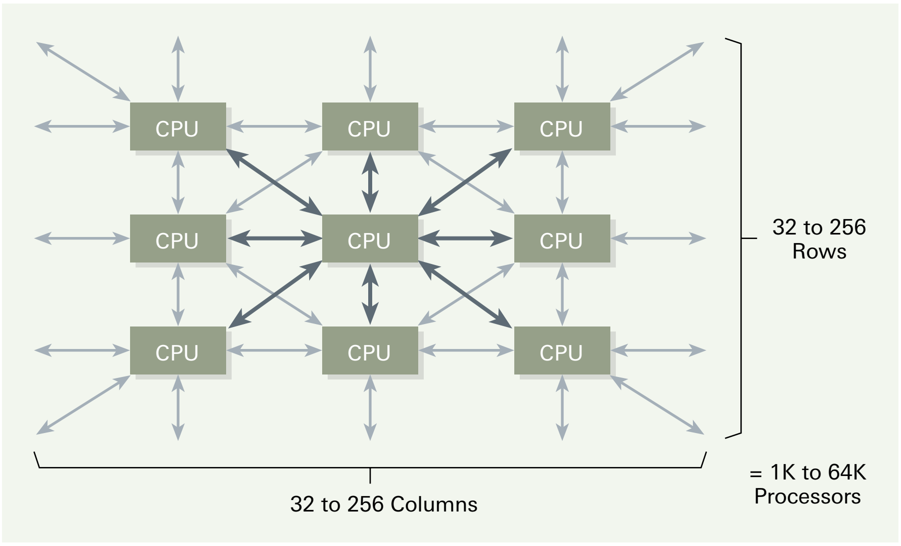


Figure 1: Massively Parallel Processor Configuration

mer passing our table asked us a question. How long would it take to generate all the values using brute force? We had a very powerful system for some kinds of problems—and mathematical computations was one of them. It would not take much programming effort to get each of the 65,536 processors to compute square roots for different values, so the computation of all 4,294,967,296 values for the 32-bit square root instruction would not take much time at all. The slowest and most difficult part of such a test would be figuring out whether or not the computed result was correct. We were also worried about common mode problems. (Those show up when the thing you're testing and the thing that generates the expected result both use the same component or algorithm. If there is an error in the shared component, both the code under test and the generated expected result will have the same wrong answer.) Scratching on the back of a napkin (literally), we figured out that even using a different algorithm on a neighboring processor was only going to take a few minutes.

We designed a test where each of the 64K CPUs started with a different value and tested each value 65,536 greater than the last one. Then, each CPU would ask a neighbor to compute the square root differently to check the original result.

Two Errors in Four Billion

Two hours later, we made the first run using our biggest processor configuration. The test and verification ran in about six minutes or so, but we were confused a bit when the test reported two values in error. Both the developer and I scratched our heads, because the two values didn't appear to be special in any way or related to one another. Two errors in four billion. The developer shook his head and said, "No…This is impossi…Oh!…Okay?" He thumbed through the microcode source listing and stabbed his finger down on an instruction. "Yeah—there it is!" There was only one thing that could cause those two values to be wrong, and a moment's thought was all it took to discover what that was!

He gleefully tried to explain that the sign on a logical micro instruction shift instruction was incorrect for a particular bit pattern on a particular processor node at 11 P.M. on the night of a full moon (or some such—it didn't make a lot of sense to me at the time—but we caught, reported, and fixed the defect, fair and square). We submitted the defect report, noting only the two actual and expected values that miscompared and attaching the test program. Then, in just a few minutes, he fixed the code and reran the test, this time with no reported errors. After that, the developers ran the test before each submission, so the test group didn't ever see any subsequent problems.

I later asked the developer if he thought the inspection we had originally planned would likely have found the error. After thinking about it for a minute, he shook his head. "I doubt if we'd have ever caught it. The values weren't obvious powers of two or boundaries or anything special. We would have looked at the instructions and never noticed the subtle error."

Be Like Don Quixote

Failures can lurk in incredibly obscure places. Sometimes the only way to catch them is through exhaustive testing. The good news was, we had been able to "exhaustively test" the 32-bit square root function (although we checked only the expected result, not other possible errors like leaving the wrong values in micro registers or not responding correctly to interrupts). The bad news was that there was also a 64-bit version, with four billion times as many values—that would take about 50,000 years to test exhaustively. Maybe between now and 50,000 years from now, we'll come up with better tests to catch these potential errors. In the meantime, I think we should continue to challenge our assumptions of what's possible in our never-ending quest to build better software. Who knows what giants we might slay? STQE

Douglas Hoffman is the principal consultant with Software Quality Methods, LLC, in Silicon Valley, providing planning, management guidance, and training to transform products and organizations. He is a Fellow of the ASQ and has earned his BACS, MSEE, and MBA degrees.

***STQE* magazine is produced by Software Quality Engineering.**

Complete coverage?

- Some people (attempt to) simplify away the problem of *complete testing* by advocating "*complete coverage*."
- What is coverage?
  - Extent of testing of certain attributes or pieces of the program, such as statement coverage or branch coverage or condition coverage.
  - Extent of testing completed, compared to a population of possible tests.
- Typical definitions are oversimplified. They miss, for example,
  - Interrupts and other parallel operations
  - Interesting data values and data combinations
  - Missing code
- The number of variables we might measure is stunning. I (Kaner) listed 101 examples in *Software Negligence & Testing Coverage.*

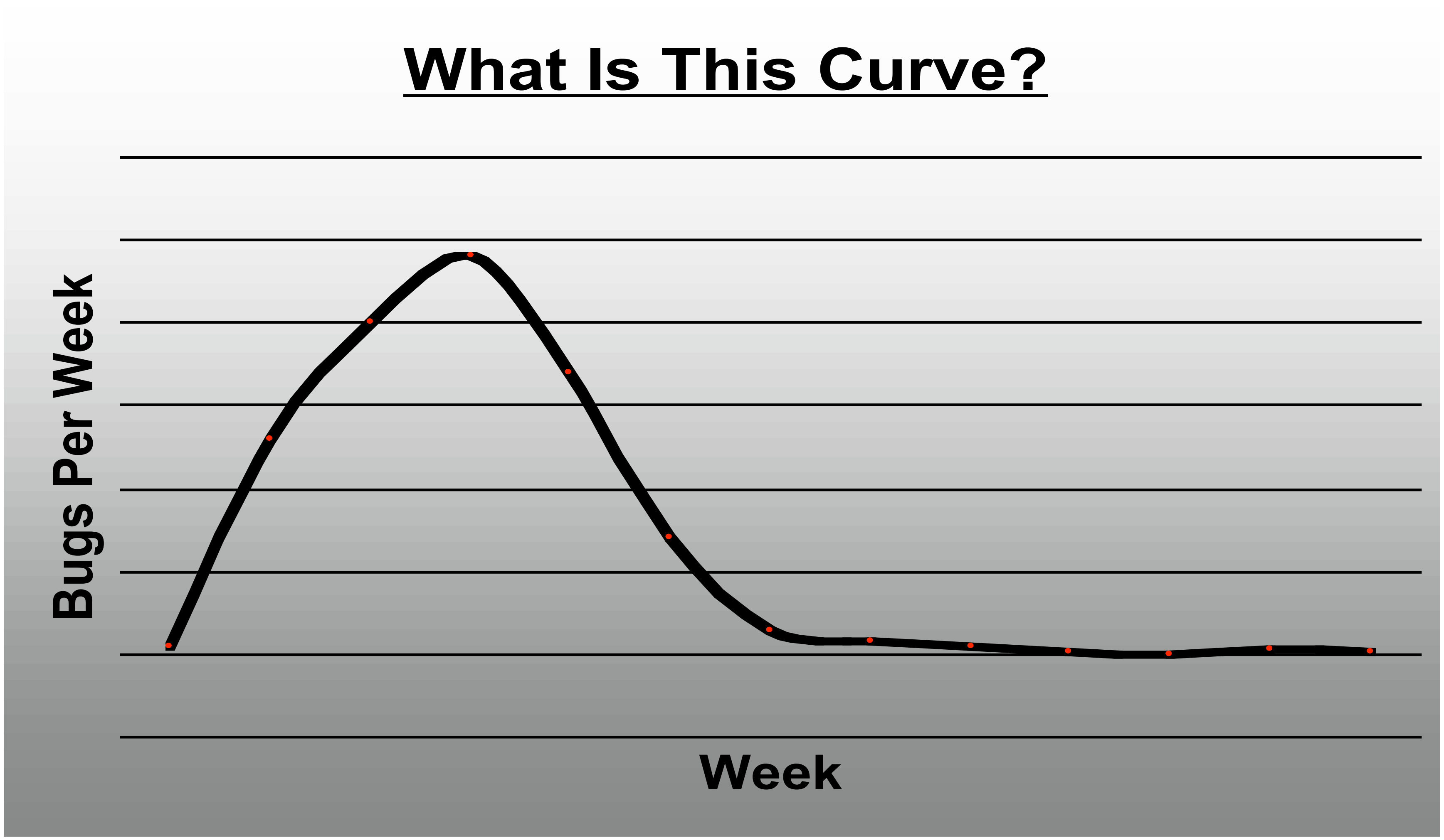
Measuring and achieving high coverage

- Coverage measurement is a useful and interesting way to tell that you are far away from complete testing.
- But testing in order to achieve “high” coverage is likely to result in development of a mass of low-power tests.
  – People optimize what we measure them against, at the expense of what we don’t measure.
    - For more on measurement distortion and dysfunction, read Bob Austin’s book, *Measurement and Management of Performance in Organizations.*
  – Brian Marick, raises this and several other issues in his papers at www.testing.com (e.g. How to Misuse Code Coverage). Brian has been involved in development of several of the commercial coverage tools.

What about low bug find rates?

- Another way people measure completeness, or extent, of testing is by plotting bug curves, such as:
  - New bugs found per week
  - Bugs still open (each week)
  - Ratio of bugs found to bugs fixed (per week)
- These curves can be useful progress indicators.
- But some people tell us to fit the data to a theoretical curve, often a probability distribution, and read our position from the curve. At some point, it is "clear" from the curve that we’re done.

The bug curve



A common model (Weibull) and its assumptions

- Testing occurs in a way that is similar to the way the software will be operated.
- All defects are equally likely to be encountered.
- All defects are independent.
- There is a fixed, finite number of defects in the software at the start of testing.
- The time to arrival of a defect follows the Weibull distribution.
- The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals.

The Weibull distribution

- I think it is absurd to rely on a distributional model (or any model) when every assumption it makes about testing is obviously false.
- One of the advocates of this approach points out that

  “Luckily, the Weibull is robust to most violations.”

  - This illustrates the use of surrogate measures—we don’t have an attribute description or model for the attribute we really want to measure, so we use something else, that is allegedly “robust”, in its place. This can be very dangerous
  - The Weibull distribution has a shape parameter that allows it to take a very wide range of shapes. If you have a curve that generally rises then falls (one mode), you can approximate it with a Weibull.

BUT WHAT DOES THAT TELL US? HOW SHOULD WE INTERPRET IT?

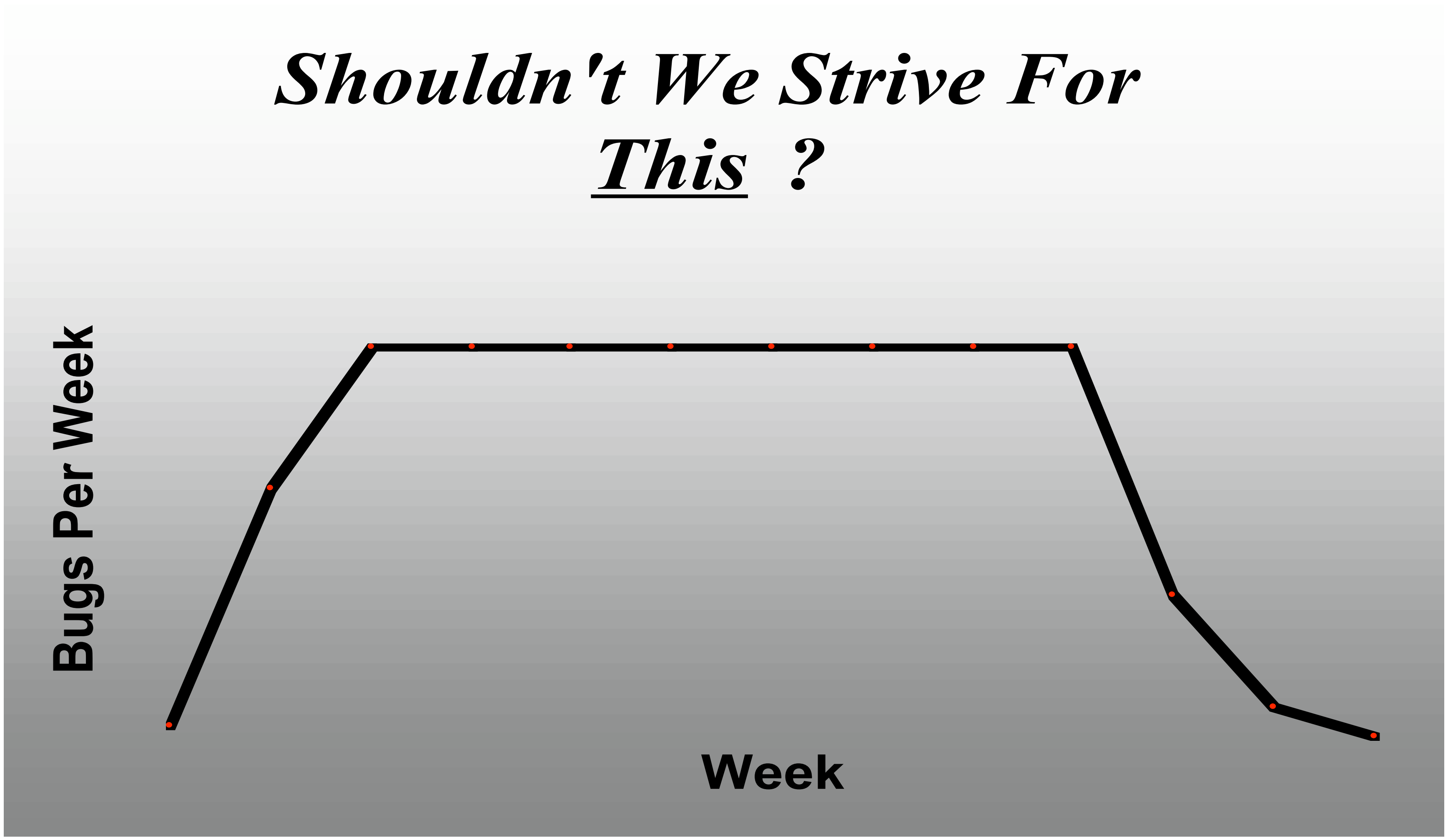
Side effects of bug curves

- Earlier in testing, the pressure is to increase bug counts. In response, testers will:
  - Run tests of features known to be broken or incomplete.
  - Run multiple related tests to find multiple related bugs.
  - Look for easy bugs in high quantities rather than hard bugs.
  - Less emphasis on infrastructure, automation architecture, tools and more emphasis of bug finding. (Short term payoff but long term inefficiency.)
- For more on measurement dysfunction, read Bob Austin’s book, *Measurement and Management of Performance in Organizations*.
- For more observations of problems like these in reputable software companies, see Doug Hoffman's article, *The Dark Side of Software Metrics.*

Side effects of bug curves

- Later in testing, there is pressure to decrease the new bug rate:
  - Run lots of already-run regression tests.
  - Don't look as hard for new bugs.
  - Shift focus to appraisal, status reporting.
  - Classify unrelated bugs as duplicates.
  - Class related bugs as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
  - Postpone bug reporting until after the measurement checkpoint (milestone). (Some bugs are lost.)
  - Report bugs informally, keeping them out of the tracking system.
  - Testers get sent to the movies before measurement checkpoints.
  - Programmers ignore bugs they find until testers report them.
  - Bugs are taken personally.
  - More bugs are rejected.

Bad models are counterproductive



Testers live and breathe tradeoffs

- When you get past the simplistic answers, you realize:

  The time needed for test-related tasks is infinitely larger than the time available.

- Example: Time you spend on

  - analyzing, troubleshooting, and effectively describing a failure

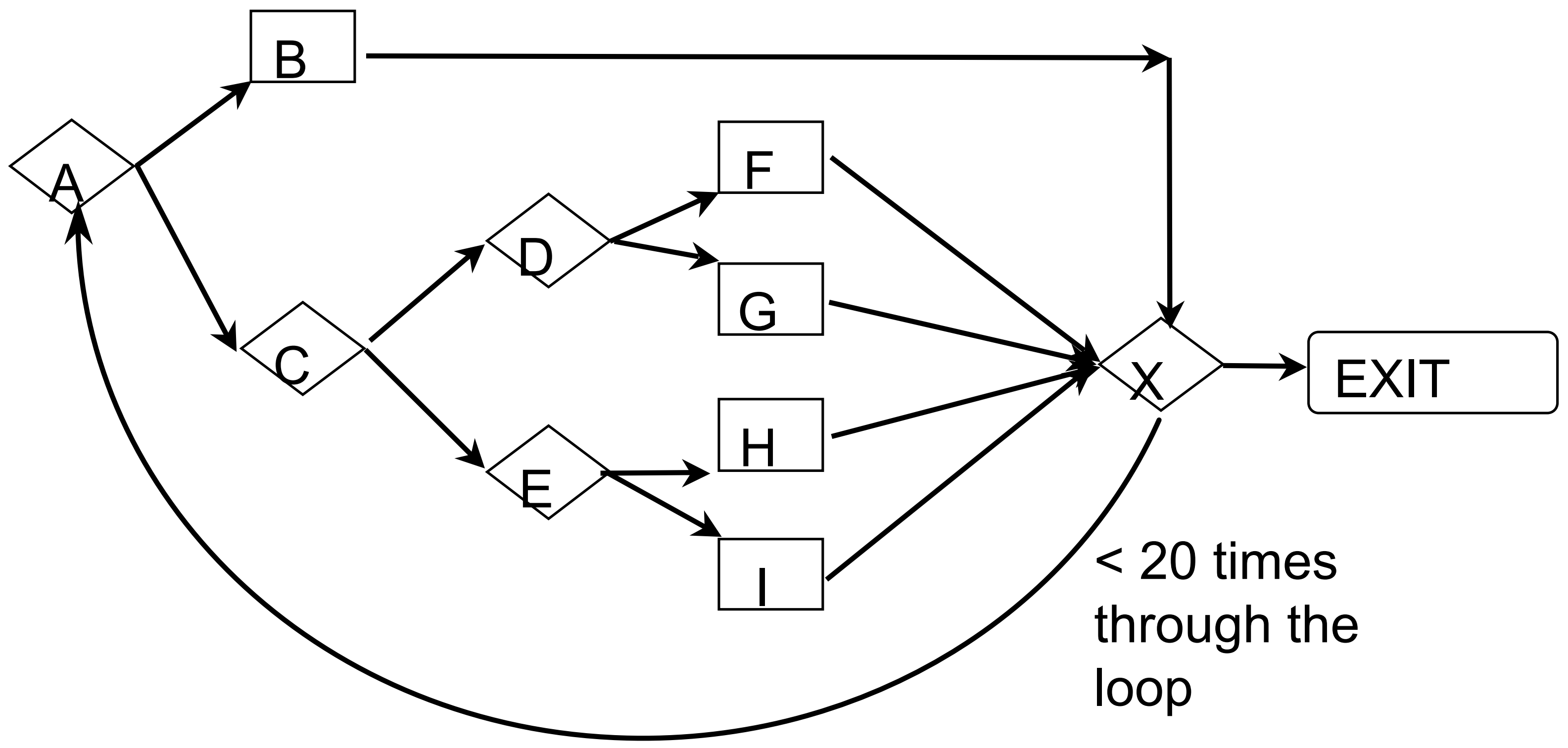
  is time no longer available for

  - Designing tests
  - Executing tests
  - Reviews, inspections
  - Retooling
  - Documenting tests
  - Automating tests
  - Supporting tech support
  - Training other staff

Combination testing

- Variables interact.
  - Example 1: a program crashed when attempting to print preview a high resolution (back then, 600x600 dpi) output on a high resolution screen. The option selections for printer resolution and screen resolution were interacting.
  - Example 2: a program fails when the sum of variables is too large.
- Suppose there are N variables.
  - Suppose the number of choices for the variables are V1, V2, through VN.
  - The total number of possible combinations is V1 x V2 x . . . x VN. This is huge.
  - For example, a field that accepts only {1, 2, 3} and another that accepts only {A, B, C} yields 9 cases, 1A, 1B, 1C, 2A, 2B, 2C, 3A, 3B, and 3C.
  - Combine two fields that accept one digit (0 to 9) each, yields 10 x 10 = 100 possible combinations.
  - There are 318,979,564,000 possible combinations of the first four moves in chess.

Sequences



Here's an example that shows that there are too many paths to test in even a fairly simple program. This is from Myers, *The Art of Software Testing.*

Sequences

The program starts at A.

- From A it can go to B or C
  - From B it goes to X
  - From C it can go to D or E
    - From D it can go to F or G
      - From F or from G it goes to X
    - From E it can go to H or I
      - From H or from I it goes to X
        - From X the program can go to EXIT or back to A. It can go back to A no more than 19 times.

One path is ABX-Exit**. There are 5 ways to get to X and then to the EXIT in one pass.**

Another path is ABXACDFX-Exit**. There are 5 ways to get to X the first time, 5 more to get back to X the second time, so there are 5 x 5 = 25 cases like this**.

Sequences

- There are $5^1 + 5^2 + ... + 5^{19} + 5^{20} = 10^{14}$ = 100 trillion paths through the program.
- Testing would take approximately one billion years to try every path (if one could write, execute and verify a test case every five minutes).

  –

Conclusion

- Complete testing is impossible
  - *There is no simple answer for this.*
  - *There is no simple, easily automated, comprehensive oracle to deal with it.*

  - *Therefore testers live and breathe tradeoffs.*

Simple Examples: Pre-tests

Suppose you were writing a text editing program that would display fonts properly on the screen. How would you test whether the font display is correct?

Here is some terminology:

A ***typeface*** is a style or design of a complete set of printable characters. For example, these are all in the typeface "Times New Roman":

Times New Roman, size 10 points	Times New Roman, size 12 points
Times New Roman, size 14 points	Times New Roman, size 16 points

The following are in the Helvetica typeface:

Helvetica, size10 points	Helvetica, size 12 points
Helvetica, size 14 points	Helvetica, size 16 points

When you add a size to a typeface, you have a font. The line above this is printed in a Helvetica 12 point font.

A How could you determine whether the editor is displaying text in the correct typeface?

B How could you determine whether the editor is displaying text in the right size?

C How would you test whether the editor displays text correctly (in any font that the user specifies)? Describe your thinking about this problem and the strategies you are considering.

Black Box Software Testing

2004 Academic Edition

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.

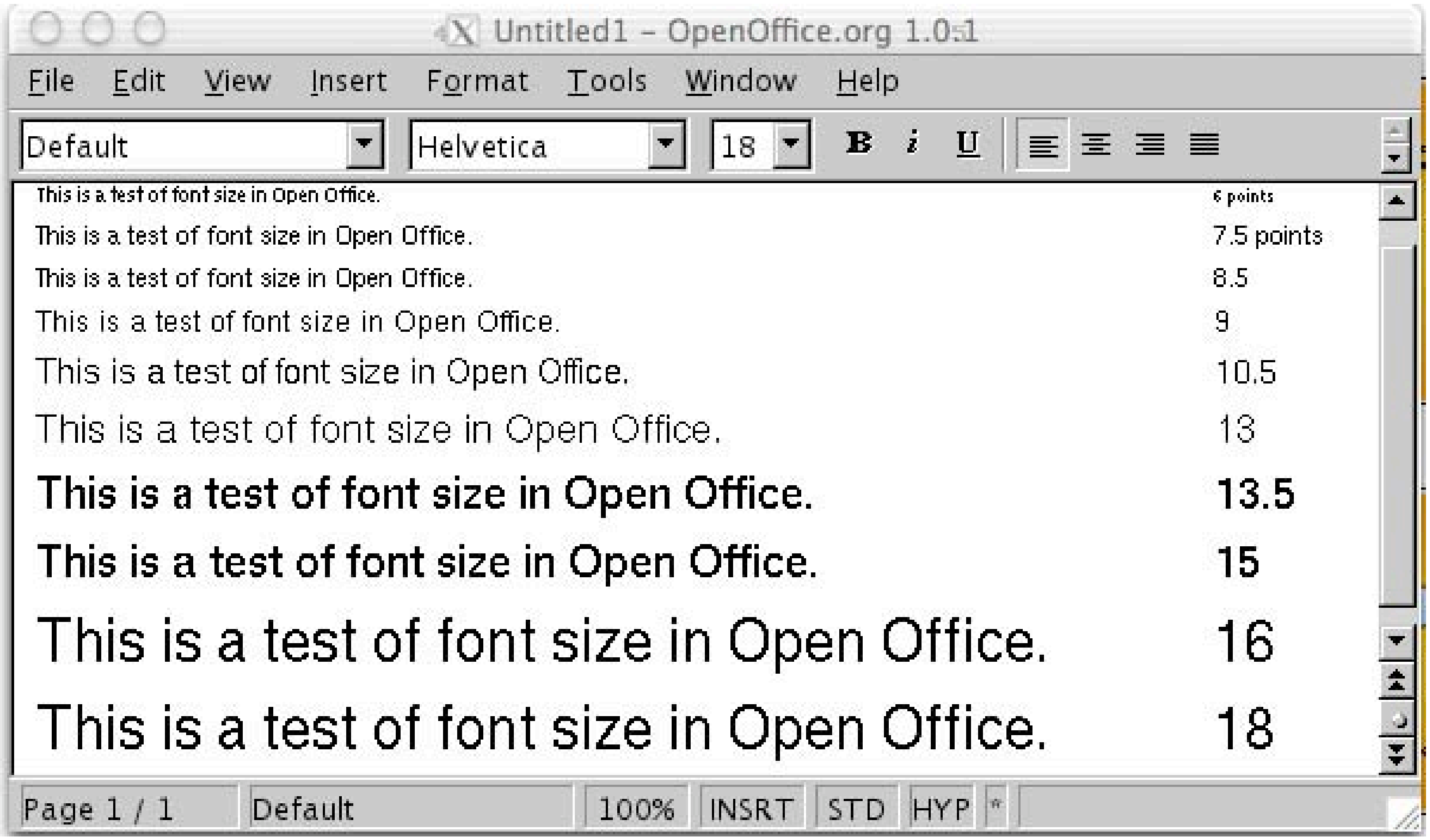
Black Box Software Testing

Part 1
Oracles

Testing is a cognitive activity,

Not a mechanical activity.

Does font size work in Open Office?


Untitled1 – OpenOffice.org 1.0.1
File Edit View Insert Format Tools Window Help
Default
Helvetica
18
B i U
Page 1 / 1
Default
100%
INSRT
STD
HYP

This is a test of font size in Open Office. 6 points
This is a test of font size in Open Office. 7.5 points
This is a test of font size in Open Office. 8.5
This is a test of font size in Open Office. 9
This is a test of font size in Open Office. 10.5
This is a test of font size in Open Office. 13
This is a test of font size in Open Office. 13.5
This is a test of font size in Open Office. 15
This is a test of font size in Open Office. 16
This is a test of font size in Open Office. 18

What's your oracle?

Oracles

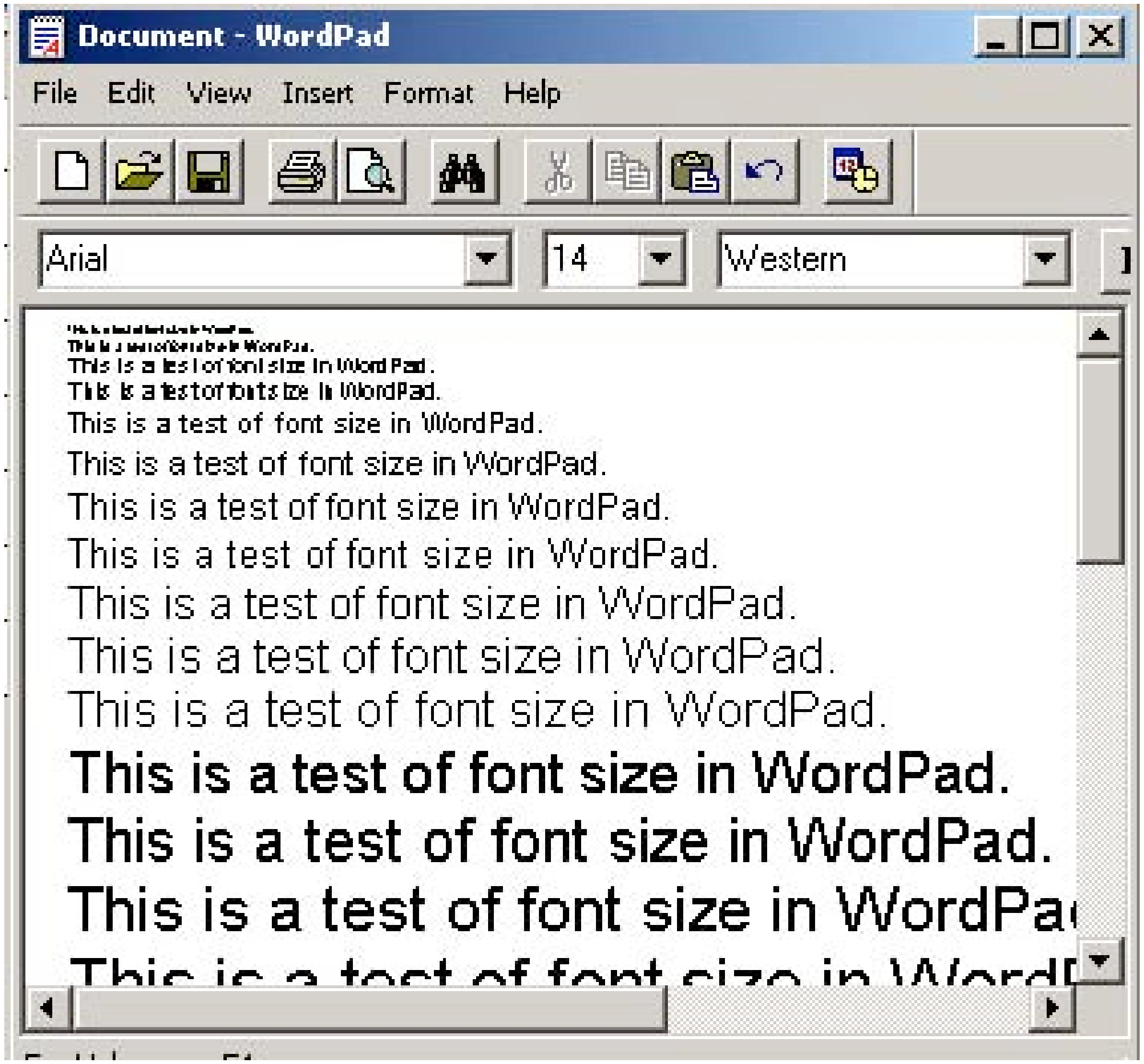
An oracle is the principle or mechanism
by which you recognize a problem.

"..it works"

really means...

*"...it appeared to meet some requirement
to some degree."*

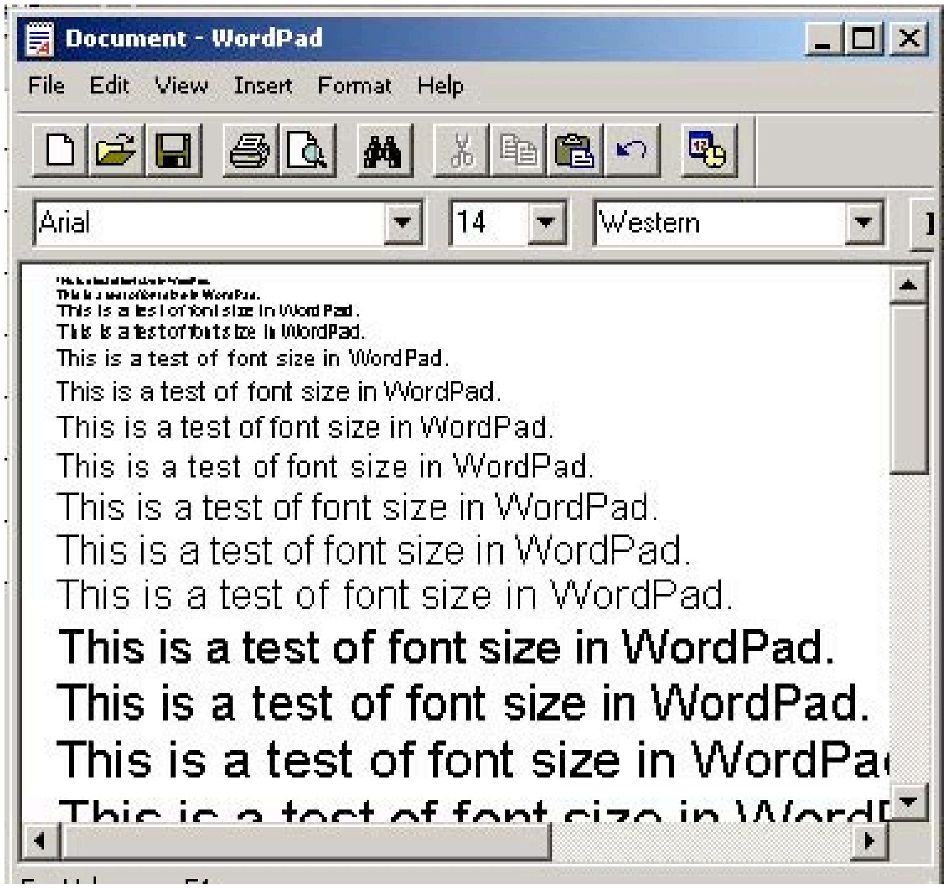
OK, so what about WordPad?



This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa
This is a test of font size in Word

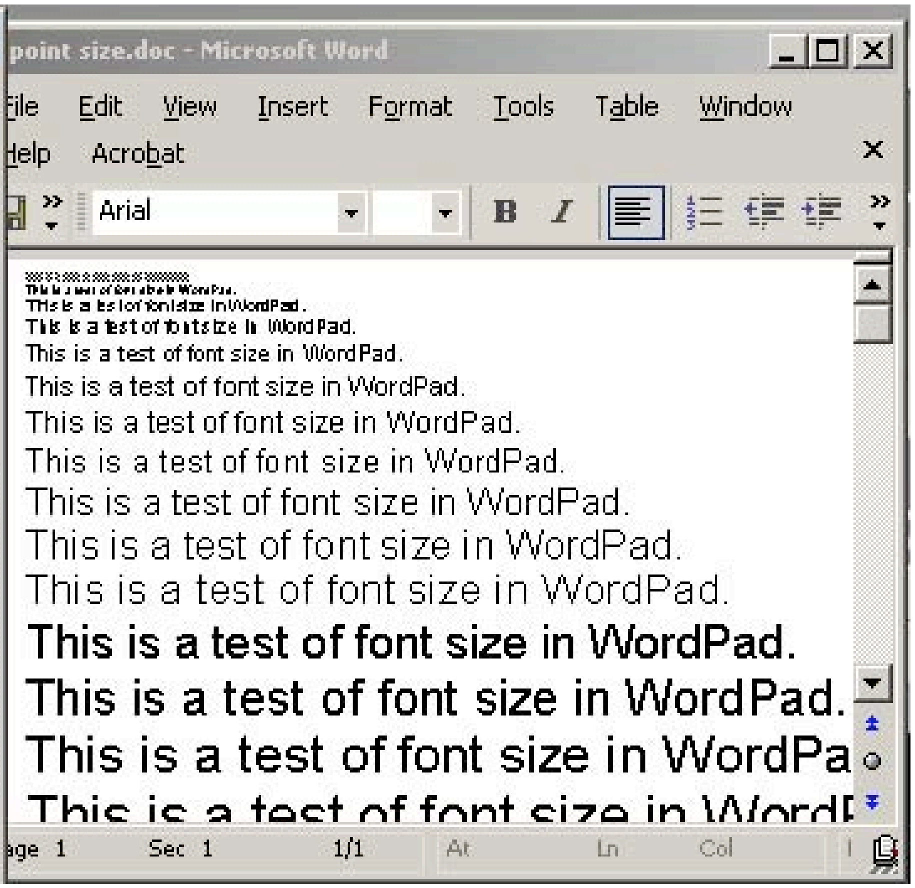
What if we compared WordPad to Word?

WordPad



This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa

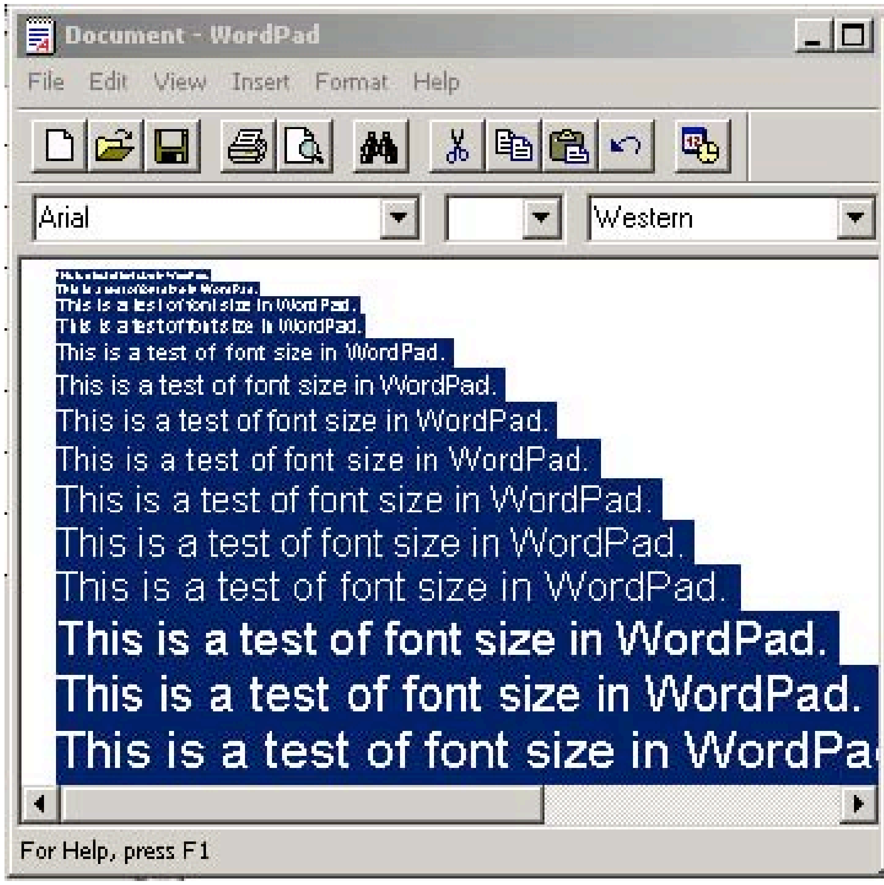
Word


point size.doc - Microsoft Word
File Edit View Insert Format Tools Table Window
Help Acrobat
Arial
age 1 Sec 1 1/1 At Ln Col

This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa

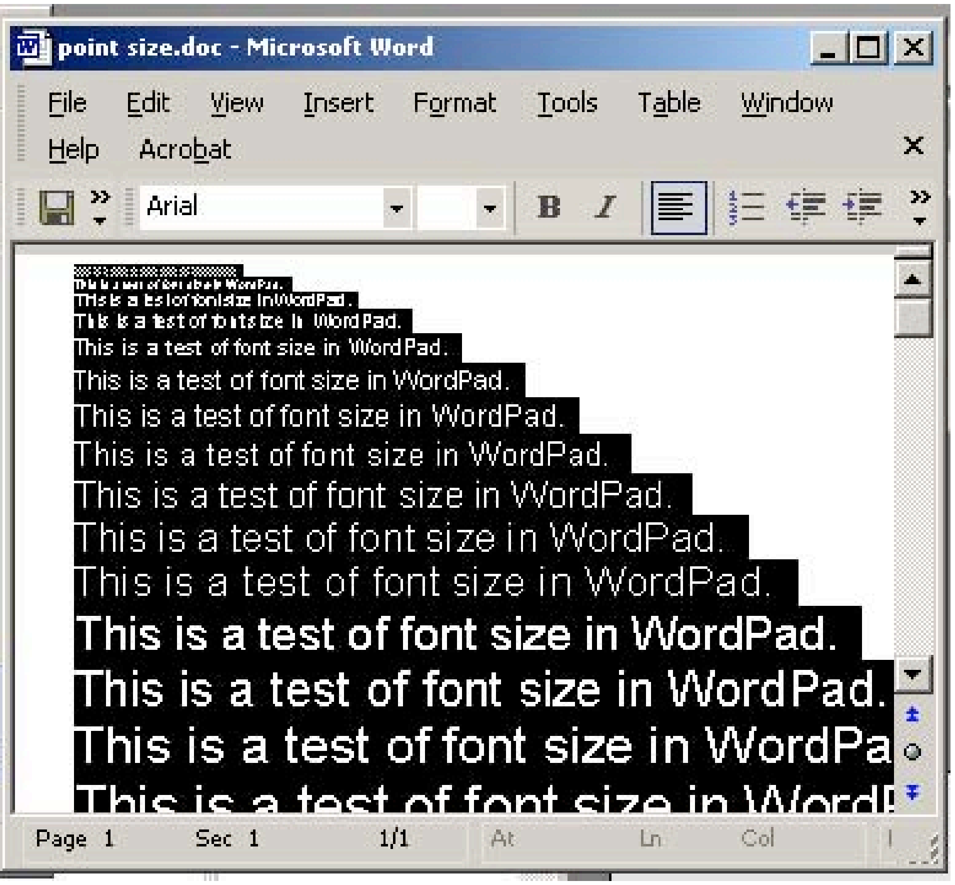
What does this tell us?

WordPad



[illegible]
[illegible]
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa

Word


point size.doc - Microsoft Word
File Edit View Insert Format Tools Table Window
Help Acrobat
Arial
Page 1 Sec 1 1/1 At Ln Col

[illegible]
[illegible]
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa
This is a test of font size in WordF

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

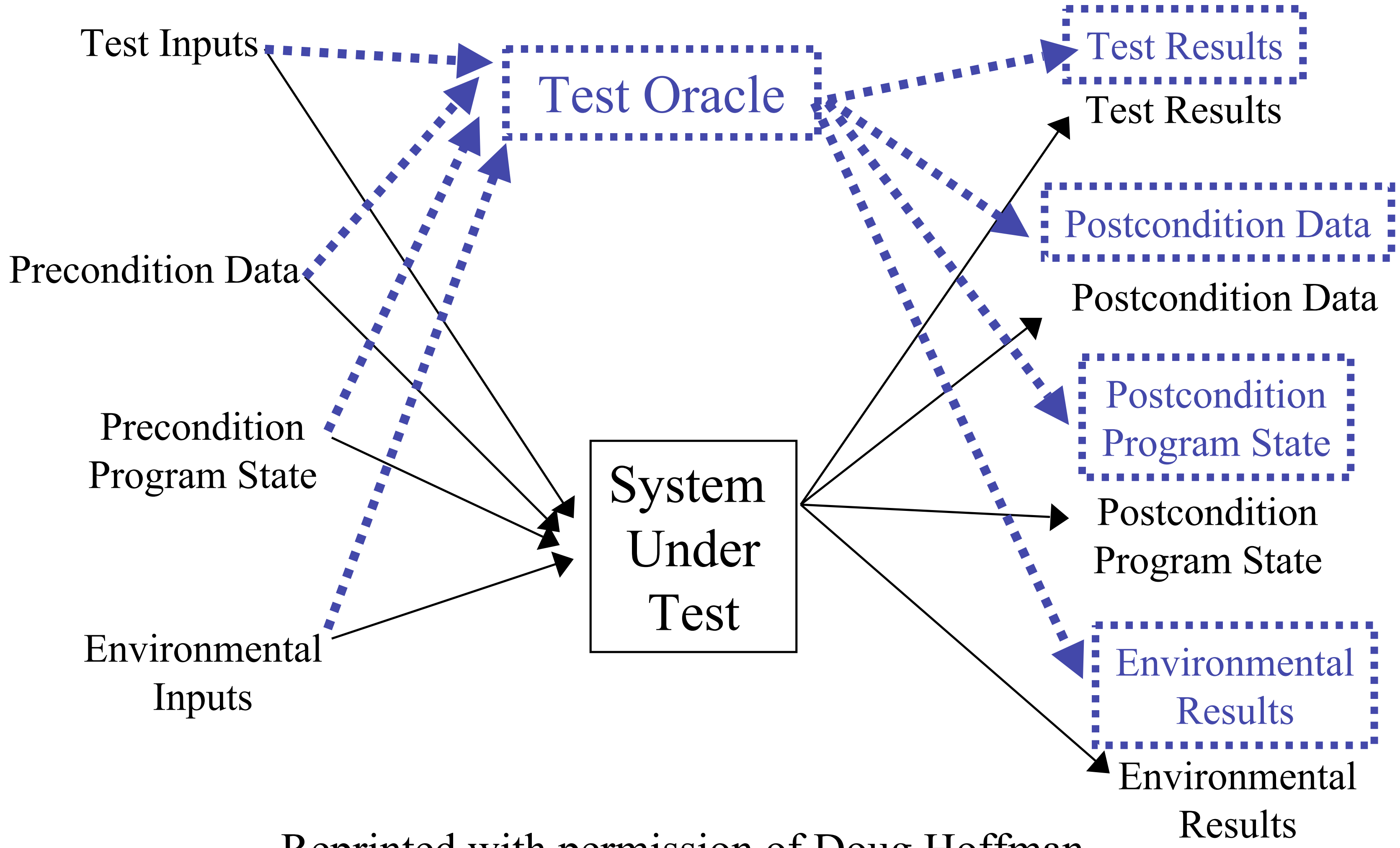
- We often hear that all testing should be automated.
- Automated testing depends on our ability to programmatically detect when the software under test fails a test.

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.

A More Complex Exercise: Open Office Test Docs.

Sun Microsystems has decided to provide financial support for development of a Release 2.0 version of Open Office.

Along with cash, they will supply staff (as they have done for some previous releases). They want to understand the commitment they should make to doing or managing the testing effort, and they have decided to retain a consulting firm to help them evaluate the effort and perhaps to help them set up the testing work or even supervise it on a long-term basis.

To select a consultant, they are running two competitions. The first competition involves many consulting firms, who are each given the same short assignment. The best four of those will be invited to California for a longer term assignment and the best one of those will get the contract.

We are in the first competition. You are a member of one of three teams being tested today. (Many other teams will be tested elsewhere, on other days.) The odds are that only one of these teams will go to California.

A More Complex Exercise: Open Office Test Docs.

Your task is to advise Sun on the type(s) and depth of test documentation needed for this project. What do they need and why do they need it?

Three executives are here from Sun today. One will go with each group, to that group's meeting room. You can ask Sun executives any questions, (they might not be able to answer some questions, but you can ask).

- After about 20 minutes, the executive will leave your room.
- After 35 minutes, you will come back to the main meeting room and each group will present its answer to the Sun executives.
- You will have 10 minutes. You can spend your 10 minutes making your presentation, answering questions, or in closing remarks after the other two groups have made their main presentation. You have 10 minutes, but only 10 minutes.

Black Box Software Testing

2004 Academic Edition

PART 21: ELEMENTS OF TEST DOCUMENATION

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.

Black Box Software Testing

Introduction to Test Documentation

Acknowledgement

- Some of the material in the sections on test documentation is from the Third Los Altos Workshop on Software Testing (LAWST), February 7 and 8, 1998. Cem founded the LAWST and was co-organizer of LAWST 3.
- At LAWST, we discussed test documentation (test planning strategies and materials). The agenda item was:
    - How do we know what test cases we have? How do we know which areas of the program are well covered and which are not?
    - How do we develop this documentation EFFICIENTLY? As many of you know, I despise thick test plans and I begrudge every millisecond that I spend on test case documentation. Unfortunately, some work is necessary. My question is, how little can we get away with, while still preserving the value of our asset?
- The following people attended LAWST 3: Chris Agruss, James Bach, Karla Fisher, David Gelperin, Kenneth Groder, Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Thanga Meenakshi, Noel Nyman, Jeffery E. Payne, Bret Pettichord, Johanna Rothman, Jane Stepak, Melora Svoboda, Jeremy White, and Rodney Wilson.
- Bret Pettichord contributed to this material as we reviewed / modified it for *Lessons Learned in Software Testing.*

Definitions

- The set of test planning documents might include:
  - A *Testing Project Plan*, which identifies classes of tasks and broadly allocates people and resources to them;
  - High-level designs for *test cases* (individual tests) and *test suites* (collections of related tests);
  - Detailed lists or descriptions of test cases;
  - Descriptions of the platforms (hardware and software environments) that you will test on, and of the relevant variations among the items that make up your platform. Examples of variables are operating system type and version, browser, printer, printer driver, video card/driver, CPU, hard disk capacity, free memory, and third party utility software.
  - Descriptions (such as protocol specifications) of the interactions of the software under test (SUT) with other applications that the SUT must interact with. Example: SUT includes a web-based shopping cart, which must obtain credit card authorizations from VISA.
  - Anything else that you would put in a hard copy or virtual binder that describes the tests you will develop and run.
- This set of materials is called the *test plan* or the *test documentation set.*

Basic Test Documentation Components

- **Lists**:
  - Such as lists of fields, error messages, DLLs
- **Outlines**: organize information into a hierarchy of lists and sublists
  - Such as a function list or an "objectives outline"
- **Tables**: organize information in two dimensions showing relationships between variables.
  - Such as boundary tables, decision tables, combination test tables
- **Matrices**: special type of table used for data collection. Rows specify what to test, columns specify the test, and you fill cells in at test time to record results.
  - Such as the numeric input field matrix, configuration matrices
- **Models**: visual, verbal, logical or tangible descriptions of the product that the tester might trace through for test ideas
  - Such as UML diagrams, architecture diagrams, state charts
    - » Refer to Testing Computer Software, pages 217-241. For more examples, see page Testing Computer Software, page 218.

Outline example: Objectives outline

- Test Objectives:
  - Inputs
    - Field-level
      - (list each variable)
    - Group-level
      - (list each interesting combination of variables)
  - Outputs
    - Field-level
      - (list each variable)
    - Group-level
      - (list each interesting combination of variables)

» (With thanks to David Gelperin, based on materials in his / SQE's *Systematic Software Testing* course)

Outline example: Objectives outline

- Requirements-based Objectives
  - Capability-based (resulting from functional design)
    - Functions or methods including major calculations (and their trigger conditions)
    - Constraints or limits (non-functional requirements)
    - Interfaces to other products
    - Input (validation) and Output conditions at up to 4 levels of aggregation, such as
      - field / icon / action / response message
      - record / message / row / window / print line
      - file / table / screen / report
      - database
    - Product states and transition paths
    - Behavior rules
      - truth value combinations

Outline example: Objectives outline

- Design-based Objectives (resulting from architectural design)
  - Processor and invocation paths
  - Messages and communication paths
  - Internal data conditions
  - Design states
  - Limits and exceptions
- Code-based Objectives
  - Control-based
    - Branch-free blocks (i.e. statements)
    - (Top) branches
    - Loop bodies: 0,1, and even
    - Single conditions: LT, EQ, and GT
  - Data-based
    - Set-use pairs
    - Revealing values for calculations

Example of a Table: Configuration Planning Table

	Var 1	Var 2	Var 3	Var 4	Var 5
Config 1	V1-1	V2-1	V3-1	V4-1	V5-1
Config 2	V1-2	V2-2	V3-2	V4-2	V5-2
Config 3	V1-3	V2-3	V3-3	V4-3	V5-3
Config 4	V1-4	V2-4	V3-4	V4-4	V5-4
Config 5	V1-5	V2-5	V3-5	V4-5	V5-5
Config 6	V1-6	V2-6	V3-6	V4-6	V5-6

This table defines 6 standard configurations for testing. In later tests, the lab will set up a Config-1 system, a Config-2 system, etc., and will do its compatibility testing on these systems. The variables might be software or hardware choices. For example, Var 1 could be the operating system (V1-1 is Win 2000, V1-2 is Win ME, etc.) Var 2 could be how much RAM on the computer under test (V2-1 is 128 meg, V2-2 is 32 meg., etc.). Var 3 could be the type of printer, Var 4 the machine's language setting (French, German, etc.). Config planning tables are often filled in using the All Pairs algorithm.

Example of a Matrix: Configuration Test Matrix

	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6
Test 1	Pass	Pass	Pass	Pass	Pass	
Test 2		Fail	Pass	Pass	Pass	
Test 3	Pass	Pass	Pass	Pass	Pass	
Test 4	Pass	Fail	Fail		Pass	
Test 5	Fail	Pass	Fail	Pass	Pass	

This matrix records the results of a series of tests against the 6 standard configurations that we defined in the Configuration Planning Table.

In this table, Config 1 has passed 3 tests, failed 1, and hasn't yet been tested with Test 2. Config 6 is still untested.

Models

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

No good test design has ever been done without models.

The trick is to become aware of how you model the product, and learn different ways of modeling.

Models and Exploration

- A model is a simplified representation of something, often something very complex, that highlights some facts or relationships (the ones the modeler thinks are important) while hiding other information.
- We usually think of modeling in terms of preparation for formal testing, but there is no conflict between modeling and exploration.
- Both types of tests start from models.
- The difference is that in exploratory testing, our emphasis is on execution (try it now) and learning from the results of execution rather than on documentation and preparation for later execution.
- UML (Unified Modeling Language) is so widely taught that we've decided to treat it as out of scope for this course. However, many testers will find a study of UML and of a modeling tool, such as Rational Rose, quite useful.
- The following, simple approaches to models, are used by several skilled testers (often to support exploratory testing) and they illustrate modeling well enough for this course's purposes.
- CAUTION: Because models hide information as well as highlighting it, they expose some issues for test analysis while obscuring others. A skilled tester will work from many different models, that highlight and hide different issues.

Architecture Diagrams

- Work from a high level design (map) of the system
  - Pay primary attention to interfaces between components or groups of components. We're looking for cracks that things might have slipped through
  - What can we do to screw things up as we trace the flow of data or the progress of a task through the system?
- You can build the map in an architectural walkthrough
  - Invite several programmers and testers to a meeting. Present the programmers with use cases and have them draw a diagram showing the main components and the communication among them. For a while, the diagram will change significantly with each example. After a few hours, it will stabilize.
  - Take a picture of the diagram, blow it up, laminate it, and you can use dry erase markers to sketch your current focus.
  - Planning of testing from this diagram is often done jointly by several testers who understand different parts of the system.

Bubble (Reverse State) Diagrams

- To troubleshoot a bug, a programmer will often work the code backwards, starting with the failure state and reading for the states that could have led to it (and the states that could have led to those).
- The tester imagines a failure instead, and asks how to produce it.
  - Imagine the program being in a failure state. Draw a bubble.
  - What would have to have happened to get the program here? Draw a bubble for each immediate precursor and connect the bubbles to the target state.
  - For each precursor bubble, what would have happened to get the program there? Draw more bubbles.
  - Keep working backwards until you reach something under your (user) control.
  - Now trace through the paths and see what you can do to force the program down one of them.

Same analysis: Fishbone diagrams

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

Bubble (Reverse State) Diagrams

- Example:
  - How could we produce a paper jam (as a result of defective firmware, rather than as a result of jamming the paper?)
  - The laser printer feeds a page of paper at a steady pace.
  - Suppose that after feeding, the system reads a sensor to see if there is anything left in the paper path. A failure could result if something was wrong with the hardware or software controlling or interpreting
    - the paper feeding (rollers, choice of paper origin, paper tray),
    - paper size,
    - clock, or
    - sensor.
  - Each of these 4 is a bubble (such as, "Something is wrong with, or interfering with, or giving bad data to, the software that interprets the sensor.") Now work backwards from that.

State Model-Based Testing

- Harry Robinson & James Tierney reported that by modeling specifications, drawing finite state diagrams of what they thought was important about the specs, or just looking at the application or the API, they could find orders of magnitude more bugs than traditional tests.
- Example, they spent 5 hours looking at the API list, found 3-4 bugs, then spent 2 days making a model and found 272 bugs. The point is that you can make a model that is too big to carry in your head. Modeling shows inconsistencies and illogicalities.
- Look at
  - all the possible inputs the software can receive, then
  - all the operational modes, (something in the software that makes it work differently if you apply the same input)
  - all the actions that the software can take.
  - Do the cross product of those to create state diagrams so that you can see and look at the whole model.
  - Use to do this with dozens and hundreds of states, Harry has a technique to do thousands of states.
    - » www.modelbasedtesting.org
    - » Paul Jorgenson, Software Testing: A Craftsman's Approach

A More Complex Exercise: Open Office Test Docs.

Group Presentations

A More Complex Exercise: Open Office Test Docs.

- After the group presentations (and critiques), I present my notes on requirements analysis for test documentation. This takes 30-45 minutes.
- The groups then meet again for 35 minutes, prepare their presentations, and then present and debrief.

Black Box Software Testing

2004 Academic Edition

PART 23 -- REQUIREMENTS ANALYSIS FOR TEST DOCUMENTATION

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.

Test documentation is a product

- How long does it take to document one test case?
  - Productivity of one hour to one day per page of test documentation?
  - If a thoroughly documented test case requires one page (or several), how many tester days (tester years) would it take to document 5000 tests?

Test documentation is an expensive product.

Test documentation is a product

- Like any expensive product, we are more likely to get a return on our investment if we understand our requirements and design / develop in order to meet those requirements.
- Prescriptive standards and templates that encourage people to do the same thing in different contexts, rather than carefully tailoring the work to the context, will yield inappropriate products (waste time and money creating the wrong types of documentation).

IEEE Standard 829 for Software Test Documentation

- Test plan
- Test-design specification
- ***Test-case specification***
  - ***Test-case specification identifier***
  - ***Test items***
  - ***Input specifications***
  - ***Output specifications***
  - ***Environmental needs***
  - ***Special procedural requirements***
  - ***Intercase dependencies***
- Test-procedure specification
- Test-item transmittal report
- Test-log

We often see one or more pages per test case.

Problems with the (allegedly) standard approach

- **High Initial Cost:** What is your documentation cost per test case?
- **High Maintenance Cost:** How many test cases will you have to modify in how many places when a change occurs?
- **Project Inertia**: The more that software design changes cause documentation maintenance costs, the more we should/will resist change.
- **Drives Test Design:** Detailed, static documentation favors invariant regression testing.
- **Discourages Exploration**: How do we document exploratory testing? If we have to document all tests, we strongly discourage exploration.
- **Discourages High Volume Automation:** How many tests can you afford to document?
- **Failing Track Record:** Many project teams start to follow 829 but then give it up mid-project. What does this do to the net quality of the test documentation and test planning effort? To legal exposure in the event of a quality-related lawsuit?

Defining documentation requirements

- Stakeholders, interests, actions, objects
  - Who would use or be affected by test documentation?
  - What interests of theirs does documentation serve or disserve?
  - What will they do with the documentation?
  - What types of documents are of high or low value?
- Asking questions
- Context-free questions
- Context-free questions specific to test planning
- Evaluating a plan

Discovering Requirements

- Requirements
  - Anything that drives or constrains design
- Stakeholders
  - Favored, disfavored, and neutral stakeholders
- Stakeholders' interests
  - Favored, disfavored, and neutral interests
- Actions
  - Actions support or interfere with interests
- Objects

Test Docs Requirements Questions

- Is test documentation a <u>product</u> or <u>tool</u>?
- Is software quality driven by <u>legal issues</u> or by <u>market forces</u>?
- How quickly is the design changing?
- How quickly does the specification change to reflect design change?
- Is testing approach oriented toward proving <u>conformance</u> to specs or <u>nonconformance</u> with customer expectations?
- Does your testing style rely more on <u>already-defined tests</u> or on <u>exploration</u>?
- Should test docs focus on <u>what</u> to test (<u>objectives</u>) or on <u>how</u> to test for it (<u>procedures</u>)?
- Should the docs ever control the testing project?

Test Docs Requirements Questions

- If the docs control parts of the testing project, should that control come early or late in the project?
- Who are the primary readers of these test documents and how important are they?
- How much traceability do you need? What docs are you tracing back to and who controls them?
- To what extent should test docs support tracking and reporting of project status and testing progress?
- How well should docs support delegation of work to new testers?
- What are your assumptions about the skills and knowledge of new testers?
- Is test doc set a process model, a product model, or a defect finder?

Test Docs Requirements Questions

- A test suite <u>should</u> provide <u>prevention</u>, <u>detection</u>, and <u>prediction</u>. Which is the most important for this project?
- How <u>maintainable</u> are the test docs (and their test cases)? And, how well do they ensure that test changes will follow code changes?
- Will the test docs help us identify (and revise/restructure in face of) a <u>permanent shift in the risk profile</u> of the program?
- Are (should) docs (be) <u>automatically created</u> as a byproduct of the test automation code?

Example: Early vs. Late Planning

- Benefits of early planning
  - Scheduling and staffing *(unless things change)*
  - Early opportunity to review *(if people can understand your work)*
- Costs of early planning
  - Any delays in finding bugs are expensive. You should start *finding* bugs ASAP.
  - Early investment in test case design is risky if the product design is subject to change.
  - You will keep learning over time. Planning early sometimes precludes groups from inventing new tests later.
  - Depth of testing should reflect risks, many of which are unclear until mid-project.
- *MY USUAL GOAL: Do just-in-time test planning without losing key benefits of early work.*

Ultimately, write a mission statement

- Try to describe your core documentation requirements in one sentence that doesn't have more than three components.

The test documentation set will primarily support our efforts to find bugs in this version, to delegate work, and to track status.

Two contrasting missions

The test documentation set will support ongoing product and test maintenance over at least 10 years, will provide training material for new group members, and will create archives suitable for regulatory or litigation use.

A More Complex Exercise: Open Office Test Docs.

Typical Student Problems in this exercise:

- They don’t realize (bwahahahaha!) that each Sun executive is different. One is President, one is VP Engineering, one is VP Marketing. We introduce the executives by name and title, but the students don’t realize that this means they will have different perspectives and contradictory requirements. As a result, they meet with only one executive.
- They don’t ask many questions, or few targeted at the test documentation requirements.
- They don’t know what to ask.
- They don’t realize that they should vary the answer according to the client’s needs.
- They don’t appreciate the complexity of Open Source development project, and call for far more paperwork than the community is likely to provide, without justifying the expense.

A More Complex Exercise: Open Office Test Docs.

Benefits:

- Lessons learned:
  - Test documentation is a product, subject to context-specific needs, wishes, and constraints.
  - Different stakeholders have different needs, wishes and constraints and often want incompatible solutions.
  - If you don't check in with your stakeholders, you're probably going to deliver the wrong thing(s).
  - There are lots of types and levels of depth / focus / scope / objectives of test documentation.
- Skills practiced
  - Oral presentation
  - Group planning
  - Requirements questioning

A More Complex Exercise: Exploration & Bug Reporting

PREFACE

- This exercise has worked well for a commercial class, whose students had used plenty of black box test techniques on the job.
- Paired exploratory testing has worked well for mixed groups (experienced paired with novice). We suspect that novice/novice pairs need introductory training in test techniques, to give them a sense of the scope open to them.
- In class, we used Open Office. Here, we're illustrating it with MS Word (because you probably know that better).
- It's important to pick a feature that students have a chance of understanding quickly. If the students are inexperienced, pick a target-rich feature, like outlining (including bullets, numbering, headings, and headings within tables).
- It's very important to assign students numbers after they split into groups. You will resplit them twice more and need to avoid prior exposures.

A More Complex Exercise: Exploration & Bug Reporting

- We're going to test the Microsoft Word's outlining feature (including bullets, numbering, headings, and headings within tables).
- To start this, we will split into pairs. Each pair will test together. Take careful notes -- in the next part of this exercise, you will enter your bug reports into the database.

A More Complex Exercise: Exploration & Bug Reporting

- **To do paired exploratory testing, start with a charter**
  - A typical exploratory testing session lasts about 60-90 minutes. Ours will run 50 minutes.
  - The charter for a session might include what to test, what risks are involved, what tools to use, what testing tactics to use, what bugs to look for, what documents to examine, what outputs are desired, etc.
    - Some testers work from a detailed project outline, picking a task that will take a day or less.
    - Others create a flipchart page that outlines this session's work or the work for the next few sessions.

A More Complex Exercise: Exploration & Bug Reporting

- A quick summary of commonly used test techniques:
  - Risk-based testing
    - Failure modes and effects testing
    - Exploratory attacks
  - Specification-based testing
  - Domain testing
  - Function testing
  - Scenario testing
  - User
  - State model
  - Regression
  - High volume automation

Black Box Software Testing

2004 Academic Edition

PART 7 -- TEST DESIGN

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.

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

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)

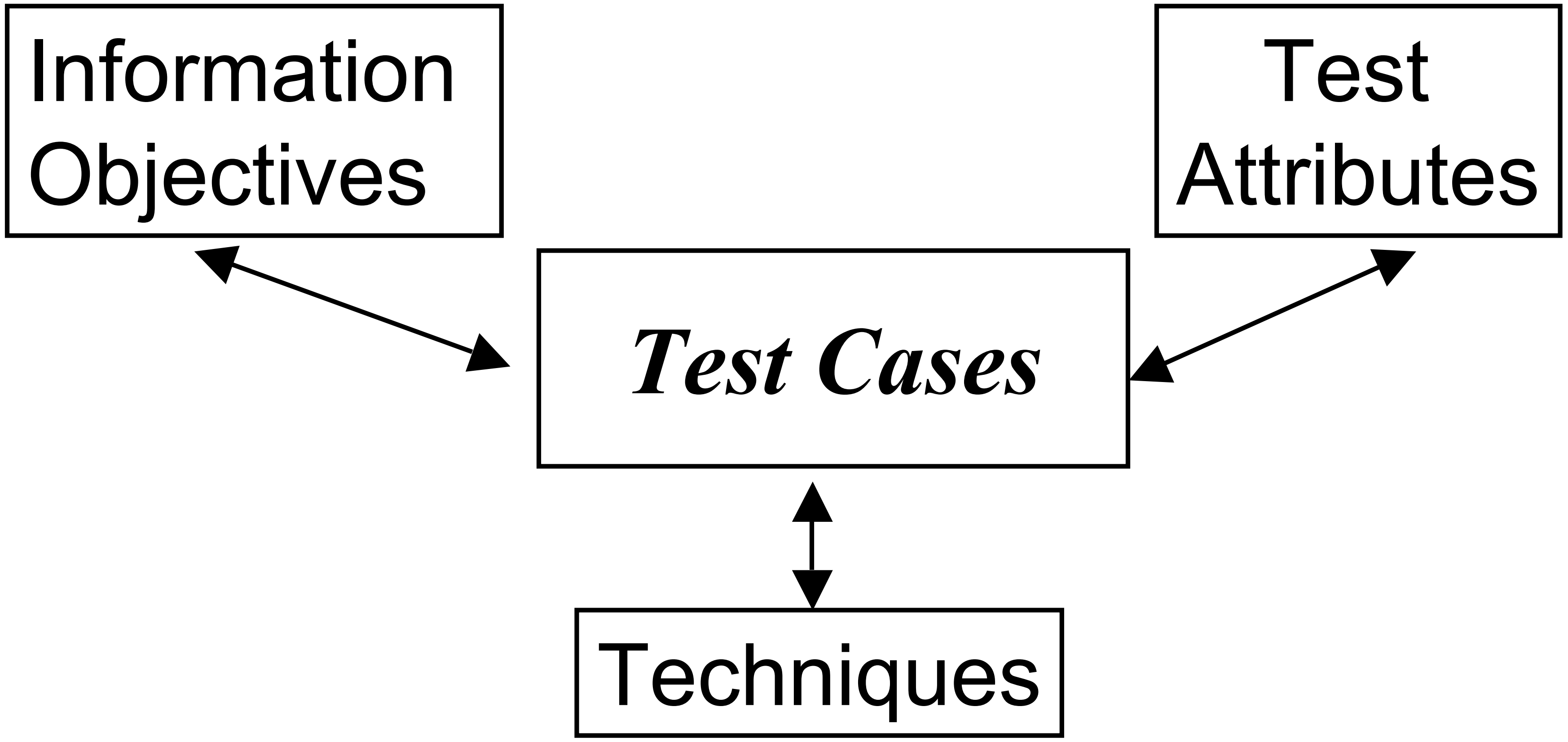
Test cases

- 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



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

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

Test Design Exercise

- Draw a table.
  - The columns are the 11 dominating test techniques (add a 12th or 13th if you have other favorites)
  - The rows are the 5 dimensions, testers, coverage, potential problems, activities, and evaluation.
  - In each cells, put an X if you think that this (column's) technique is primarily focused on that (cell's) dimension.
- Below the table (or on another page):
  - For each technique
    - Start a new paragraph
    - Write no more than 100 words
    - Briefly define or describe the technique
    - Explain why you think the technique is primarily focused on the dimensions you selected

Test Design Exercise

- Draw a table.
  - The columns are the 11 dominating test techniques (add a 12th or 13th if you have other favorites)
  - The rows are the 9 attributes: *power, significance, credibility, representativeness, easy to evaluate, useful for troubleshooting, informative, appropriate complexity, trigger*s *insight*
  - In each cell, put an X if you think that this (column's) technique is primarily focused on that (cell's) dimension.
- Below the table (or on another page):
  - For each technique
    - Start a new paragraph
    - Write no more than 100 words
    - Briefly define or describe the technique
    - Explain why you think the technique is primarily focused on the dimensions you selected

A More Complex Exercise: Exploration & Bug Reporting

TESTING

TESTING

TESTING

A More Complex Exercise: Exploration & Bug Reporting

NOW LET'S REPORT THE BUGS

1. Find a new partner
2. Each of you should pick your two favorite bugs (from the ones you found)
3. For each bug, write a report that has two parts
   - Summary: a brief (one-line) description of the problem
   - Problem & How to Reproduce It
     - Describe the problem in a persuasive way (your report will, or will not, convince someone to spend time fixing this bug).
     - Describe the problem clearly--step by step, numbered steps, describing exactly how to reproduce the bug
4. Review your partner's bug reports and suggest improvements; fix your reports

A More Complex Exercise: Exploration & Bug Reporting

EVALUATING THE BUGS

You're now going to find yet another new partner and evaluate some bug reports. Before we do the evaluation, here are some guidelines for figuring out what is a good bug report.

- Bug advocacy
  - making bugs more compelling
    - clear statement of worst consequence
    - follow-up tests if useful
    - scope identified and clear
  - eliminating objections
    - reproducibility
    - lack of clarity (or just plain unintelligible)
    - insult
- Bug evaluation (see slides)

Software Testing SWE 5411

Assignment #1

Replicate and Edit Bugs

Editing Bugs

The purpose of this assignment is to give you experience editing bugs written by other people. This task will give you practice thinking about what a professional report should be, before you start entering your own reports into this public system.

- Work with Mozilla Firebird, version 0.7.
- Read the instructions at http://bugzilla.mozilla.org/enter_bug.cgi and http://www.mozilla.org/quality/help/beginning-duplicate-finding.html and http://bugzilla.mozilla.org/queryhelp.cgi#status. Read the bug entry guidelines at http://bugzilla.mozilla.org/docs/html/Bugzilla-Guide.html#BUG_WRITING.
- Find 5 bug reports in IssueZilla about problems with Mozilla Firebird that appear to have not yet been independently verified. These are listed in the database as "unconfirmed". As of February 3, 2004, there appear to be 876 such reports.
- For each report, review and replicate the bug, and add comments *as appropriate* in the Additional Comments field to the report on issuezilla.
- The assignment that you submit *to me* should list the bug numbers. I will read the comments you filed (if any) on the bug report. In addition, for each bug, tell me what was done well, what was done poorly and what was missing that should have been there in the bug report.

Editing Bugs Assignment Procedure

For each bug:

- Review the report for clarity and tone (see "first impressions", next slide).
  - » Include comments on clarity and tone on the memo you send me (but don't make these comments on the bug report itself)
- Attempt to replicate the bug.
  - » Send comments to me on the replication steps (were the ones in the report clear and accurate), your overall impressions of the bug report as a procedure description, and describe any follow-up tests that you would recommend.
- You may edit the bug report yourself, primarily in the following ways.
  - » Add a comment indicating that you successfully replicated the bug on XXX configuration in YYY build.
  - » Add a comment describing a simpler set of replication steps (if you have a simpler set). Make sure these are clear and accurate.
  - » Add a comment describing why this bug would be important to customers (this is only needed if the bug looks minor or like it won't be fixed. It is only useful if you know what you are talking about).
  - » Your comments should NEVER appear critical or disrespectful of the original report or of the person who wrote it. You are adding information, not criticizing what was there.
- If you edit the report in the database, **never change what the reporter has actually written**. You are not changing his work, you are adding comments to it at the end of the report
- Your comments should have your name and the comment date, usually at the start of the comment, for example: "(Cem Kaner, 12/14/01) Here is an alternative set of replication steps:")

Editing Bugs—First impressions

- Is the summary short (about 50-70 characters) and descriptive?
- Can you understand the report?
  - As you read the description, do you understand what the reporter did?
  - Can you envision what the program did in response?
  - Do you understand what the failure was?
- Is it obvious where to start (what state to bring the program to) to replicate the bug?
- Is it obvious what files to use (if any)? Is it obvious what you would type?
- Is the replication sequence provided as a numbered set of steps, which tell you exactly what to do and, when useful, what you will see?

Editing Bugs—First impressions

- Does the report include unnecessary information, personal opinions or anecdotes that seem out of place?
- Is the tone of the report insulting? Are any words in the report potentially insulting?
- Does the report seem too long? Too short? Does it seem to have a lot of unnecessary steps? (This is your first impression—you might be mistaken. After all, you haven’t replicated it yet. But does it LOOK like there’s a lot of excess in the report?)
- Does the report seem overly general (“Insert a file and you will see” – what file? What kind of file? Is there an example, like “Insert a file like blah.foo or blah2.fee”?)

Editing Bugs—Replicate the Report

- Can you replicate the bug?
- Did you need additional information or steps?
- Did you get lost or wonder whether you had done a step correctly? Would additional feedback (like, “the program will respond like this...”) have helped?
- Did you have to guess about what to do next?
- Did you have to change your configuration or environment in any way that wasn’t specified in the report?
- Did some steps appear unnecessary? Were they unnecessary?
- Did the description accurately describe the failure?
- Did the summary accurate describe the failure?
- Does the description include non-factual information (such as the tester’s guesses about the underlying fault) and if so, does this information seem credible and useful or not?

Editing Bugs—Follow-Up Tests

- Are there follow-up tests that you would run on this report if you had the time?
  - *In follow-up testing, we vary a test that yielded a less-than-spectacular failure. We vary the operation, data, or environment, asking whether the underlying fault in the code can yield a more serious failure or a failure under a broader range of circumstances.*
  - *You will probably NOT have time to run many follow-up tests yourself. For evaluation, my question is not what the results of these tests were. Rather it is, what follow-up tests should have been run—and then, what tests were run?*
- What would you hope to learn from these tests?
- How important would these tests be?

Editing Bugs—Follow-Up Tests

- Are some tests so obviously likely to yield important information that you feel a competent reporter would have run them *and described the results?*
  - The report describes a corner case without apparently having checked non-extreme values.
  - Or the report relies on other specific values, with no indication about whether the program just fails on those or on anything in the same class (what is the class?)
  - Or the report is so general that you doubt that it is accurate ("Insert any file at this point" – *really? Any file? Any type of file? Any size? Did the tester supply* reasons for you to believe this generalization is credible? Or examples of files that actually yielded the failure?)

Editing Bugs—Tester's evaluation

- Does the description include non-factual information (such as the tester’s guesses about the underlying fault) and if so, does this information seem credible and useful or not?
- Does the description include statements about why this bug would be important to the customer or to someone else?

The report need not include such information, but if it does, it should be credible, accurate, and useful.

GRADING NOTES FOR THE BUG EDITING ASSIGNMENT

Two components for grading the papers –

1) Comments at Bugzilla
2) Editor's report submitted to us.

I allocate 14 points possible for each bug, but total out of 10. That is, if you get a 3/14 for the bug, your score is 3/10. Similarly, 14/14 becomes 10/10. There are 10 points available for each bug.

COMMENTS ON THE BUG REPORTS THEMSELVES, FILED IN ISSUEZILLA

The content of your comments must vary depending on the problem. The key thing is that the follow-up report has to be useful to the reader.

For example, a simple failure to replicate might be sufficient (though it is rarely useful unless it includes a discussion of what was attempted.) Sometimes, detailed follow-up steps that simplify or extend the report are valuable.

This is worth up to 7 points out of 10

	Subcomponents of the comments at Issuezilla	**Points possible**
1	Report states configuration and build	+ Up to 1
2	If the report is disrespectful in tone, zero the grade for the report.	0 for the report
3	If you clearly report a simpler set of replication steps	+ Up to 5
4	If you clearly report a good follow-up test	+ Up to 5
5	A follow-up test or discussion that suggests to me that you don't understand the bug is not worth much.	+ Up to 1
6	If there is enough effort and enough usable information in the follow up test.	+ Up to 3
7	If you make a good argument regarding importance (pro or con)	+ Up to 5
8	If the bug is in fact not reproducible, and the report demonstrates that you credibly tested for reproducibility	+ Up to 5
9	You report a failure to reproduce a bug on alternate configuration without discussion	- 1
10	You report a failure to reproduce a bug on alternate configuration that was already dismissed	- 2

<u>REPORT TO ME</u>

This is worth up to 7 points out of 10

Here, you evaluate the report rather than trying to improve it. I want to see details that suggest you had insight into what makes bug reports good or bad, effective or ineffective. I did not expect you to walk through every item in the checklist and tell me something for each item (too much work, most of it would have wasted your time). Instead, I expected that you would raise a few issues of interest and handle them reasonably well. For different reports, you might raise very different issues.

1) I am interested in comments on:

   a) What was done well.
   b) What was done poorly.
   c) What was missing that should have been there.

2) In the assignment, the checklist suggests a wide range of possible comments, on

   a. First impressions
   b. Replication
   c. Follow-up tests
   d. Closing impressions

The comments do not have to be ordered in any particular way but they should address the issues raised in the assignment notes in a sensible order. I credit them as follows:

> Individual issue discussions are worth up to 3 points, but are normally worth 0.5 or 1 point (typically 1 point if well done). An exceptional discussion that goes to the heart of the quality of the report or suggests what should have been done in a clear and accurate way is worth 2 points. An exceptional and extended (long) discussion that goes to the heart of the quality of the report AND includes follow-up test discussion or suggests (well) what should should have been done is worth 3 points.

3) The primary basis of the evaluation in this section is insight into the quality of the bug report. If the student has mechanically gone through the list of questions, without showing any insight, the max point count is 5. If I see insight, the max point count is 7. A discussion that shows that the tester did not understand the bug under consideration is worth at most 5, and usually worth less.

Bug number	**Comments at issuzilla**	**Editor's report**	**Total points**
1	**7**	**7**	**10**
2	**7**	**7**	**10**
3	**7**	**7**	**10**
4	**7**	**7**	**10**
5	**7**	**7**	**10**
GRAND TOTAL			**50**

FINAL GRADING = GRAND TOTAL * 2 / 10

Instructor notes

First pairs

ab	cd	ef	gh

Second pairs

ac	bd	eg	fh

Third pairs

ae(gi)	bf(hj)	cg(km)	dh(ln)

- Pair ae(gi): Person (a) and Person (e) jointly review the bug reports submitted by person (g) and person (i).
- Note that (a) and (e) have not worked together and that neither (a) nor (e) has previously seen bugs from (g) or (i).

A More Complex Exercise: Exploration & Bug Reporting

EVALUATE BUGS

1. Pick a new partner (see instructor notes). You will role play.
   - One of you is the test manager. You want bugs reported competently, and you want bugs fixed.
   - One of you is the project manager. You expect bug reports to assist, and never interfere, with your staff's efficience in evaluating and fixing bugs. You want to ship the product on time, with the right features, within budget, and in all important respects, working.
2. Use the attached notes on evaluating bugs as a guide, a set of (all of them optional) ideas for evaluating the quality of the bug report.
3. For each bug, evaluate the quality of the bug report
4. Evaluate the significance of the bug
5. ON COMPLETION OF THE TASK, THE GROUP DEBRIEFS, DISCUSSING THE BUG REPORTS AND THE DYNAMICS OF REVIEWING THEM.

A More Complex Exercise: Exploration & Bug Reporting

- Challenges:
  - this is a long series of tasks
  - this requires prep lectures for students who have no background
  - this requires lab space for paired work
- What is learned:
  - role playing
  - communication skills
  - bug analysis skills
  - bug reporting skills
  - timeboxed exploratory testing
  - paired testing
  - chartering

Black Box Software Testing

2004 Academic Edition

PART 18 -- PAIRED EXPLORATORY TESTING

by

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

Professor of Software Engineering

Florida Institute of Technology

and

James Bach

Principal, Satisfice Inc.

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

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

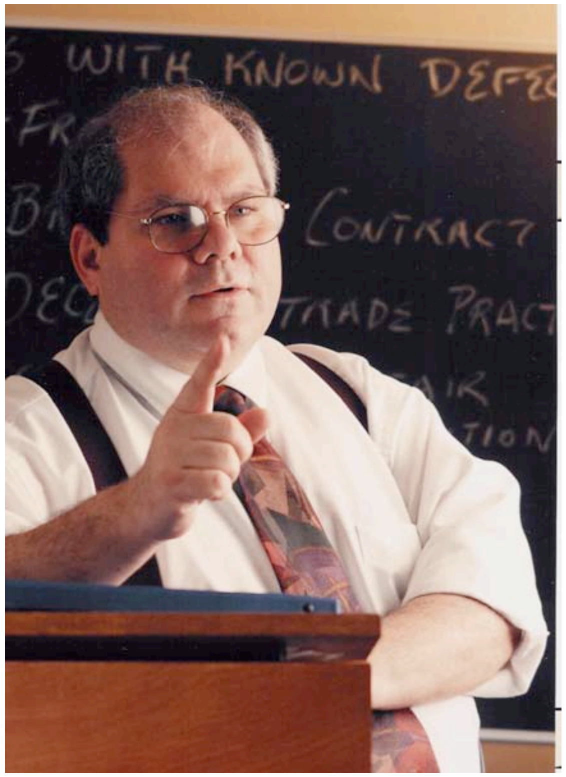
- Paired testing is *not* a vehicle for fobbing off errand-running on a junior tester. The pairs are partners, the junior tester is often the one at the keyboard, and she is always allowed to try out her own ideas.
- Accountability must belong to one person. Beck and Jeffries, et al. discuss this in 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.

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.



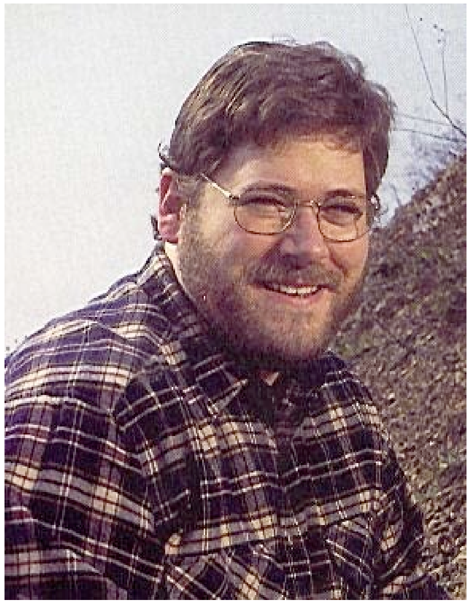
I've worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality. These have provided many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

Notice

- These notes were originally developed in co-authorship with Hung Quoc Nguyen. James Bach has contributed substantial material. I also thank Jack Falk, Elizabeth Hendrickson, Doug Hoffman, Bob Johnson, Brian Lawrence, Melora Svoboda, 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.
- These notes include some legal information, but you are not my legal client. I do not provide legal advice in the notes or in the course. Even if you ask me a question about a specific situation, you must understand that you cannot give me enough information in a classroom setting for me to respond with a competent legal opinion. I may use your question as a teaching tool, and answer it in a way that I believe would "normally" be true but my answer may be inappropriate for your particular situation. I cannot accept any responsibility for any actions that you might take in response to my comments in this course. If you need legal advice, please consult your own attorney.
- The practices recommended and discussed in this course are useful for an introduction to testing, but more experienced testers will adopt additional practices. I am writing this course with the mass-market software development industry in mind. Mission-critical and life-critical software development efforts involve specific and rigorous procedures that are not described in this course.