Developing Skills as an Exploratory Tester

Quality Assurance Institute
November 2006

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

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.

Much of the material in these slides was provided or inspired by James Bach, Michael Bolton, Jonathan Bach, and Mike Kelly. Andy Tinkham contributed significantly to the planning and design of this tutorial.

Session Blurb

Software testing is an empirical, technical investigation that we conduct to provide stakeholders with information about the quality of the product or service under test.

Exploratory testing is a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design and test execution as mutually supportive activities that run in parallel throughout the project.

My goal in this tutorial is to help you identify and develop some of the skills involved in effective exploratory testing.

Today's Agenda

- What do *you* think exploratory testing is?
- What do you see as the advantages of exploratory testing?
- What do you see as disadvantages of exploratory testing?
- What do you see as the risks of exploratory testing?
- What skills do you think are important for exploratory testing?

My intent is to focus today's tutorial on the issues you raise, rather than following a preset patter.

- I have a context-setting introduction, and then we'll revisit the questions above.
- I really want to do some work on concept mapping as a support for modeling
- Beyond that, the slides that follow are a support structure for the agenda we choose, not the agenda we will follow.

Overview

"REQUIRED" MATERIAL

- A crisis in software testing
- Updating the practice of software testing
- Overview of Exploratory testing
- Analyzing a sample requirements specification
- Using the Satisfice Heuristic Test Strategy Model to guide analysis
- Using and developing models in software testing

THE PREPARED SEQUENCE

- From information to test: Using quicktests
- The challenge of relevance
- An overview of test techniques
- Scenario testing: Using stories as a vehicle for achieving relevance

THERE'S PLENTY OF ADDITIONAL MATERIAL ON THE DISK

The current crisis in software testing

A crisis in software testing?

- NIST

  The Economic Impacts of Inadequate Infrastructure for Software Testing www.nist.gov/director/prog-ofc/report02-3.pdf

- Hmmm …
  - Software defects cost the economy a lot of money
  - Let's blame that on bad testing
    - > To use these numbers for testing, we ignore the other (non-testing) factors (e.g. organizational) that lead to product release with known defects or non-surprising defects
  - I suppose we could use that as a basis for declaring a testing "crisis" and selling lots of test consulting services

That's not the "crisis" I'm talking about today.

Here's the crisis on my radar

- We have gotten much better at
  - testing
  - documenting the testing
  - reporting status of the testing

  of small programs.

- But as the size of programs grows geometrically
  - and the efficiency of testers grows maybe linearly
    - > we impact less of the program every year.

System-level testing will become irrelevant, because we will impact so little of the product.

The traditional approach

Define system level software testing as

- functional,
- focusing on verification of the program's features,
- preferably against an authoritative specification.

The traditional approach is

- Easy to understand.
- Easy to translate into low-skill work and routine automation.
- NOT what you want as your career path.

AND

- Maybe not so effective
- Maybe outdated

An underlying crisis

Most of today's software testing techniques were developed in the 1970's.

- Back then, long programs were 10,000 statements
- Code was often readable COBOL
- An enterprising tester could read the entire program, identify all the variables and most of the relevant combinations.

An underlying crisis

We have not experienced revolutions in testing practice. We are not much more productive today than we were three decades ago:

- Regression test automation offers small, incremental improvements in productivity
- High-volume test automation is still rarely done and is poorly understood by the general (e.g. academic) testing community
- Test case documentation is as overblown as ever, with a new generation of semi-automated "test-case management" bureaucracy to slow us down further.
- Look at the *National Defense Industrial Association's Systems Engineering Workshop on the Top 5 Software Issues*, Washington, D.C., August 2006. https://acc.dau.mil/GetAttachment.aspx?id=119000&pname=file&aid=24945

Another Traditional Focus of Testing:

Find Software Errors

But an error:

- May or may not be a coding error
- May or may not be a functional error

> ***The tester who looks only for coding errors misses all of the other ways in which the program is of lower quality than it should be.***

(This was accepted by most good testing practitioners that I knew as far back as 1983.)

A slightly less traditional view

So what does the future look like?

- Five trends seem obvious to me:
  - Context-awareness in test planning ("no best practices")
  - Test-driven programming or other extensive unit testing
  - Test design readily derived from state models (or other automatable models)
  - High-volume test automation
  - Exploratory testing
- I think these are important but insufficient
  - These are reactions / extensions to 1980's style testing
  - The next generation (reactions / extensions to modern practice instead of 1980's practice) will develop new paradigms. Our task is to update their foundations education, to help them get past where we are stuck.

Basic Background -- Quality and errors

Quality is value to some person

-- Jerry Weinberg

Under this view:

- Quality is inherently subjective
  - The value differs from person to person and therefore so does the quality

Software error

An attribute of a software product

- that reduces its value to a favored stakeholder
- or increases its value to a disfavored stakeholder
- without a sufficiently large countervailing benefit.

- In other words:

"A bug is something that bugs somebody."

James Bach

Reject the "Not My Job" definition of testing

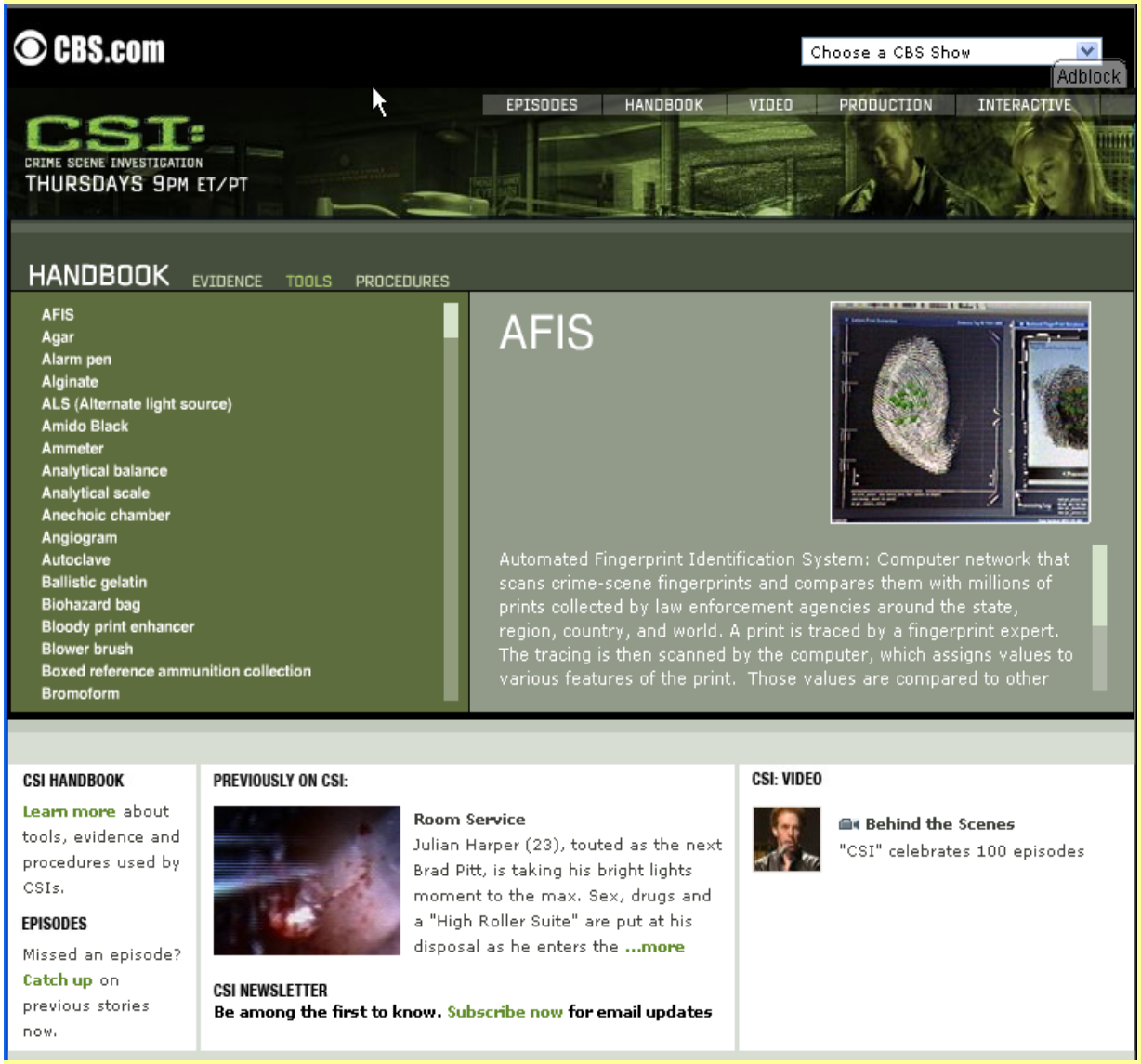
- Testing is **not only** about finding errors in code.
- Testing is **not only** about doing tasks that some programmer can imagine for you or meeting the objectives that some programmer wishes on you
  - *unless that programmer is your primary stakeholder*
- Anything that threatens value of a product to a stakeholder with influence threatens quality in a way important to the project.
  - *You might be called on to investigate any of those possible threats, including security, performance, usability, suitability for intended purpose, etc.*

Tasks beyond your personal skill set can still be within your scope.

Software testing

- is an empirical
- technical
- investigation
- conducted to provide stakeholders
- with information
- about the quality
- of the product or service under test

It's kind of like CSI



MANY tools, procedures, sources of evidence.

- Tools and procedures don't define an investigation or its goals.
- There is too much evidence to test, tools are often expensive, so investigators must exercise judgment.
- The investigator must pick what to study, and how, in order to reveal the most needed information.

Defining Testing

Empirical

- "derived from experiment and observation rather than theory."

technical

- We use technical means, including experimentation, logic, mathematics, models, tools (testing-support programs), and tools (measuring instruments, event generators, etc.)

investigation

- An organized and thorough search for information.
- This is an active process of inquiry. We ask hard questions (aka run hard test cases) and look carefully at the results.

Defining Testing

Empirical technical investigation …

that we conduct to provide stakeholders

- Someone who has a vested interest in the success of the product
- Someone who has a vested interest in the success of the testing effort
- A "stakeholder with influence" is someone who has authority to influence the design or marketing of the product.

with information

- The information of interest is often about the presence (or absence) of bugs.
- Quality-revealing information apart from specific bugs may be more vital to a particular stakeholder at a particular time

Examples of Information Objectives

- Find important bugs, to get them fixed
- Assess the quality of the product
- Help managers make release decisions
- Block premature product releases
- Help predict and control product support costs
- Check interoperability with other products
- Find safe scenarios for use of the product
- Assess conformance to specifications
- Certify the product meets a particular standard
- Ensure the testing process meets accountability standards
- Minimize the risk of safety-related lawsuits
- Help clients improve product quality & testability
- Help clients improve their processes
- Evaluate the product for a third party

Different objectives require different testing strategies and will yield different tests, different test documentation and different test results.

Defining Testing

Empirical technical investigation that we conduct to provide stakeholders with information…

<u>about the quality</u>

- Quality is value to some person. (Weinberg's definition)
  - Note that this is inherently subjective. The quality of an item differs from person to person.
  - Anything that reduces the value of the product to a stakeholder is a quality-related issue.
  - Testers look for different things, for different stakeholders

Defining Testing

Empirical technical investigation of the product under test that we conduct to provide stakeholders with information about the quality …

<u>of the product or service under test</u>

- The product includes the data, the documentation, the hardware, whatever the customer gets. If it doesn't all work together, it doesn't work.
- The product is a solution space to a set of problems. If it doesn't solve the problem(s), it doesn't work.

Testing is always done within a context

- Testing is done in the face of harsh constraints
  - Complete testing is impossible
  - The project schedule and budget are finite
  - The skills of the testing group are limited
- Testing is done on behalf of stakeholders
  - Project manager, marketing manager, customer, programmer, competitor, attorney
  - Which stakeholder(s) ***this time***?
    - > What information are they interested in?
    - > What risks are they trying to mitigate?
- Testing might be done before, during or after a release.
- Improvement of product or process might or might not be an objective of testing.

Example of context: A thought experiment

Suppose you were testing a program that does calculations, like a spreadsheet. Consider 4 development contexts:

1. Computer game that uses the spreadsheet for occasional tasks like bargaining with another player
2. Early development of a commercial product, at the request of the project manager, to help her identify product risks and help her programmers understand the reliability implications of their work
3. Late development of a commercial product, to help the project manager decide whether the product is finished
4. Control the operation of medical equipment or collect and store the results of research on the operational safety of the equipment.

A thought experiment (slide 2)

For each context:

- What is your mission?
- How could you organize testing to help you achieve the mission?
  - How aggressively should you hunt for bugs? Why?
  - Which bugs are less important than others? Why?
  - How important are issues of performance (speed of operation)? Polish of the user interface? Precision of the calculations? Prevention and detection of tampering with the data?
  - How extensively will you document your work? Why?
  - What other information would you expect to provide to the project (if any)? Why?

Examples of important context factors

- Who are the stakeholders with influence
- What are the goals and quality criteria for the project
- What skills and resources are available to the project
- What is in the product
- How it could fail
- Potential consequences of potential failures
- Who might care about which consequence of what failure
- How to trigger a fault that generates the failure we're seeking
- How to recognize failure
- How to decide what result variables to pay attention to
- How to decide what *other* result variables to pay attention to in the event of intermittent failure
- How to troubleshoot and simplify a failure, so as to better
  - motivate a stakeholder who might advocate for a fix
  - enable a fixer to identify and stomp the bug more quickly
- How to expose, and who to expose to, undelivered benefits, unsatisfied implications, traps, and missed opportunities.

The Analogy to Manufacturing QC

- Fixed design
- Well understood risks
- The same set of errors appear on a statistically understood basis
- Test for the same things on each instance of the product
- Scripting makes a lot of sense

The Analogy to Design QC

- The design is rich and not yet trusted
- A fault affects every copy of the product
- The challenge is to find new design errors, not to look over and over and over again for the same design error

- Scripting is probably an industry worst practice for design QC

Software testing is assessment of a design, not of the quality of manufacture of the copy

Imagine ...

Imagine crime scene investigators

- (real investigators of real crime scenes)
- following a script.

How effective do you think they would be?

What we need for design...

Is a constantly evolving set of tests

- That exercise the software in new ways (new combinations of features and data)
- So that we get our choice of
  - broader coverage of the infinite space of possibilities
    - > adapting as we recognize new classes of possibilities
  - and sharper focus
    - > on risks or issues that we decide are of critical interest today.

For that

we do

exploratory testing


Exploratory testing

Exploratory software testing

- is a style of software testing
- that emphasizes the personal freedom and responsibility
- of the individual tester
- to continually optimize the value of her work
- by treating
  - test-related learning,
  - test design,
  - test execution, and
  - test result interpretation
- as mutually supportive activities
- that run in parallel throughout the project.

Exploratory testing

- **Learning**: Anything that can guide us in what to test, how to test, or how to recognize a problem.
- **Design:** "to create, fashion, execute, or construct according to plan; to conceive and plan out in the mind" (Websters)
  - Designing is not scripting. The representation of a plan is not the plan.
  - Explorers' designs can be reusable.
- **Execution:** Execution can be automated or manual.
- **Interpretation:** What do we learn from program as it performs under our test
  - about the product and
  - about how we are testing the product?

Exploratory testing

- **Learning**: Anything that can guide us in what to test, how to test, or how to recognize a problem, such as:
  - the **project context** (e.g., development objectives, resources and constraints, stakeholders with influence), **market forces** that drive the product (competitors, desired and customary benefits, users), hardware and software **platforms**, and **development history** of prior versions and related products.
  - **risks, failure history, support record** of this and related products and how this product currently behaves and fails.

Examples of learning activities

- **Study competitive products** (how they work, what they do, what expectations they create)
- **Research the history** of this / related products (design / failures / support)
- **Inspect the product under test** (and its data) (create function lists, variable lists, data relationship charts, file structures, user tasks, product benefits, FMEA)
- **Question**: Identify missing info, imagine potential sources and potentially revealing questions (interview developers, users, and other stakeholders, fill in or supplement answers from reference materials)
- **Review written sources:** specifications and other authoritative documents, culturally authoritative sources, persuasive sources
- **Try out potentially useful tools**

Examples of learning activities (continued)

- **Create and apply models**: state, usage, data, data flow, other relationships/dependencies among data, task workflows, user expectations, physical systems or business systems being automated or simulated
- **Hardware / software platform:** Design and run experiments to establish lab procedures or polish lab techniques. Research the compatibility space of the hardware/software (see, e.g. Kaner, Falk, Nguyen's (Testing Computer Software) chapter on Printer Testing).
- **Team research:** brainstorming or other group activities to combine and extend knowledge
- **Paired testing:** mutual mentoring, foster diversity in models and approaches.

Examples of design activities

Design: "to create, fashion, execute, or construct according to plan; to conceive and plan out in the mind" (Websters)

- Map test ideas to FMEA or other lists of variables, functions, risks, benefits, tasks, etc.
- Map test techniques to test ideas
- Map tools to test techniques.
- Map staff skills to tools / techniques, develop training as necessary
- Develop supporting test data
- Develop supporting oracles
- Data capture: notes? Screen/input capture tool? Log files? Ongoing automated assessment of test results?
- Charter: Decide what you will work on and how you will work

Examples of execution activities

- Configure the product under test
- Branch / backtrack: Let yourself be productively distracted from one course of action in order to produce an unanticipated new idea.
- Alternate among different activities or perspectives to create or relieve productive tension
- Pair testing: work and think with another person on the same problem
- Vary activities and foci of attention
- Create and debug an automated series of tests
- Run and monitor the execution of an automated series of tests

Interpretation activities

- Part of interpreting the behavior exposed by a test is determining whether the program passed or failed the test.
- A mechanism for determining whether a program passed or failed a test is called an **oracle**. We discuss oracles in detail, on video and in slides, at http://www.testingeducation.org/BBST/BBSTIntro1.html
- Oracles are heuristic: they are incomplete and they are fallible. One of the key interpretation activities is determining which oracle is useful for a given test or test result:

Interpretation activities: Examples of oracles

- **Consistent within Product:** Behavior consistent with behavior of comparable functions or functional patterns within the product.
- **Consistent with Comparable Products:** Behavior consistent with behavior of similar functions in comparable products.
- **Consistent with a Model's Predictions:** Behavior consistent with expectations derived from a model.
- **Consistent with History**: Present behavior consistent with past behavior.
- **Consistent with our Image:** Behavior consistent with an image that the organization wants to project.
- **Consistent with Claims:** Behavior consistent with documentation or ads.
- **Consistent with Specifications or Regulations:** Behavior consistent with claims that must be met.
- **Consistent with User's Expectations:** Behavior consistent with what we think users want.
- **Consistent with Purpose:** Behavior consistent with apparent purpose.

A different structuring of key activities

Jonathan Bach, Mike Kelly, and James Bach are working on a broad listing / tutorial of ET activities which I hope to see in book form.

See Bachs' presentation on Exploratory Testing Dynamics at http://www.quardev.com/whitepapers.html

We reviewed preliminary drafts of Exploratory Testing Dynamics at the Exploratory Testing Research Summit (spring 2006) and Consultants Camp 2006 (August), looking specifically at teaching issues.

The four-page handout that I provide here provides an outline for what should be a 3-4 day course. It's a stunningly rich set of skills. I hope to get a chance to take a Bach / Kelly course on ET Dynamics in the near future.

In this abbreviated form, the lists are particularly useful for audit and mentoring purposes, to help you spot gaps in your test activities or those of someone whose work you are evaluating.

Analyzing a sample requirements specification

The specification

"The Disaster Missing Person Tracker Website"

- Anonymized (and slightly revised) student project
- Developed in a requirements course by a team of grad students with significant development experience

The opening exercise with this specification

- Please review the specification, working in groups of 2 to 4.
- Please imagine that this is a genuine document, that it has gone through its approval process, and that you are now analyzing the document from the point of view of how you will test the product, rather than how you want someone else to revise the specification.
- As you sample the document, please consider:
  - What tests (clusters of tests) should be run for a given requirement?
  - How much more (or what instead) is needed compared to the tests provided
  - If you had the code in front of you, would tests of the code NOW help clarify the specification?
  - What key information is missing and how would you get it?

Notes on spec-based testing from Kaner & Bach's BBST course

We've seen at least three different meanings of specification-based testing

- **A style of testing (collection of test-related activities and techniques) focused on discovering what claims are being made in the specifications and on testing them against the product.**

  This is what we mean by spec-based testing.

- A style of testing focused on proving that the statements in a specification (and the code that matches the statements) are logically correct.
- A set of test techniques focused on logical relationships among variables that are often detailed in specifications.

Context factors

- Is this *intended* as an authoritative document? Who is its champion?
- Who cares if it's kept up to date and correct? Who doesn't?
- Who is accountable for its accuracy and maintenance?
- What are the corporate consequences if it is inaccurate?

Why did they write the specification?

- Enforceable contract for custom software?
- Facilitate and record agreement among stakeholders? About specific issues or about the whole thing?
- Vision document?
- Support material for testing / tech support / technical writers?
- Marketing support?
- Sales or marketing communication?
- Regulatory compliance?

Context factors

- To what extent is a test against the spec necessary, sufficient, or useful?
- To what extent can you change the product or process via spec review / critique?
- Will people invest in your developing an ability to understand the spec?

Why are you reviewing the spec or testing the product against the specification?

- Contract-related risk management?
- Regulatory-related risk management?
- Development group wants to use the spec as an internal authoritative standard?
- Learn about the product?
- Prevent problems before they are coded in?
- Identify testing issues before you get code?
- Help company assess product drift?
- It's a source of information—test tool to help you find potential bugs? (in product or spec?)

Spec testing issues

What **is** the specification?

What does the specification say?

Critiquing the specification (what it says):

- How it says what it says
- What it says about the product
- What it says about the testing of the product

Critiquing the specification (doing the critique)

Driving tests from the specification

Legal issues

What *is* the specification?

What is **a** specification?

- For our purposes, we include any document that describes the product and drives development, sale, support, or the purchasing of the product.

What is the scope of **this** specification?

- Some specs cover the entire product, others describe only part of it (such as error handling).
- Some specs address the product from multiple points of view, others only from one point of view.

Do we have the **right** specification?

- Right version?
- Source control?
- Do we verify version?
  - File compares?

What *is* the specification?

Is this a stable specification?

- Is it under change control?
  - Should it be?

Supplementary information assumed by the specification writer

- Some aspects of the product are unspecified because they are defined among the staff, perhaps in some other (uncirculated?) document

Implicit specifications

- Some aspects of the product are unspecified because there are controlling cultural or technical norms.
- These are particularly important
  - Rather than making an unsupported statement that "It's bad" (e.g. "users won't like it"), you can justify your assertions

Implicit specifications

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

Implicit specifications

- Interview development staff from the last version.
- Look at customer call records from the previous version. What bugs were found in the field?
- Usability test results
- Beta test results
- 3rd party tech support databases, magazines and web sites with reports of bugs in your product, common bugs in your niche or on your platform and for discussions of how some features are supposed (by some) to work.
- Get lists of compatible equipment and environments from Marketing (in theory, at least.)
- Localization guide (probably published for localizing products on your platform.)
- Look at compatible products, to find their failures (then look for these in your product), how they designed features that you don't understand, and how they explain their design. See listservs, websites, etc.
- Exact comparisons with products you emulate
- Content reference materials (e.g. an atlas to check your on-line geography program)

Spec testing issues

What is the specification?

What does the specification say?

Critiquing the specification (what it says):

- How it says what it says
- What it says about the product
- What it says about the testing of the product

Critiquing the specification (doing the critique)

Driving tests from the specification

Legal issues

What does the spec say?

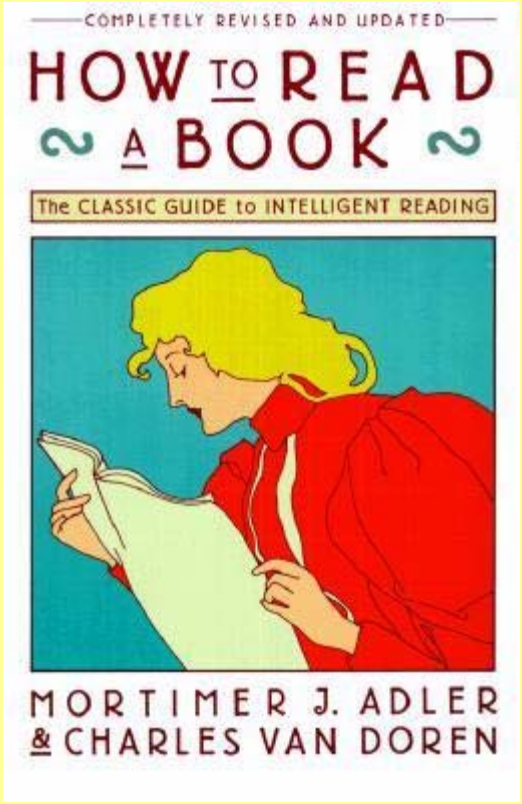
Much of what is written about specification analysis has to do with the specification-in-the-small—interpreting the fine details in one or two pages of text

- These are useful skills
- But specifications are often one or two *thousand* pages (or more)
  - spread across multiple documents
  - which incorporate several other documents by reference
  - using undefined, inconsistently defined or idiosynchratically defined vocabulary

Specification readers often suffer severe information overload.

Active reading skills and strategies are essential for effective specification analysis

Basics of active reading



Adler, M.J. and van Doren, C. (1972) How to Read a Book.

http://radicalacademy.com/adlermarkabook.htm

http://www.justreadnow.com/strategies/active.htm

http://www.somers.k12.ny.us/intranet/reading/PLAN.html

http://www.mindtools.com/pages/article/newISS_04.htm

http://www.clt.astate.edu/bdoyle/TextbookRdng.ppt

http://titan.iwu.edu/~writcent/Active_Reading.htm

http://istudy.psu.edu/FirstYearModules/Reading/Materials.html

http://www.itrc.ucf.edu/forpd/strategies/stratCubing.html

http://www.ncrel.org/sdrs/areas/issues/students/learning/lr2befor.htm

Active reading

Prioritize what you read, by

- **Surveying (read table of contents, headings, abstracts)**
- **Skimming (read quickly, for overall sense of the material)**
- **Scanning (seek specific words or phrases)**

Search for information in the material you read, by

- **Asking information-gathering questions and search for their answers**
- **Creating categories for information and read to fill in the categories**
- **Questioning / challenging / probing what you're reading**

Organize it

- **Read with a pen in your hand**
- **If you underline or highlight, don't do so until AFTER you've read the section**
- **Make notes as you go**
  - Key points, Action items, Questions, Themes, Organizing principles
- **Use concise codes in your notes (especially on the book or article). Make up 4 or 5 of your own codes. These 2 are common, general-purpose:**
  - ? means I have a question about this
  - ! means new or interesting idea
- **Spot patterns and make connections**
  - Create information maps
- **Relate new knowledge to old knowledge**

Plan for your retention of the material

- **SQ3R (survey / question / read / recite / review)**
- **Archival notes**

Active Reading: Cubing

Cubing involves attacking a problem from 6 perspectives. Originally developed as a writing strategy, it's often suggested for active reading as well.

For the feature or concept that you are trying to understand:

1. **Describe it:** describe its physical attributes (size, shape, etc.) and its functional attributes;
2. **Compare it:** What's it similar to? Why do you think so?
3. **Associate it:** What other ideas, products, etc. does it bring to mind?
4. **Analyze it:** Break it down into its components. How are they related? How do they work together?
5. **Apply it:** What can you (or the user) do with it?
6. **Evaluate it:** Take a stand. List reasons that it is good (good feature, good implementation, good design, good idea, etc.) or bad. If you want to be neutral, make two lists—one of all the ways that it's good, the other of all the ways that it's bad.

As you develop your cube, work through the specification (and any other documents you have) to collect the information you need to do these tasks.

Asking questions

Here are some key contrasts:

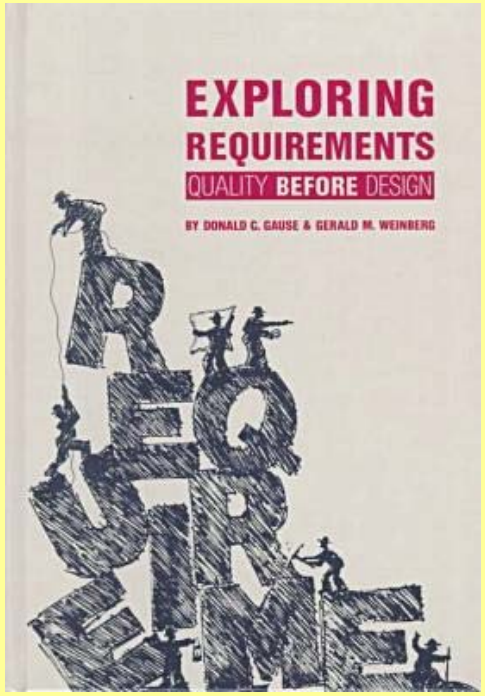
Hypothetical *(what would happen if …)* **vs. behavioral** *(what have you done / what has happened in the past in response to …)*

Factual *(factual answers can be proved true or false)* **vs. opinion** *(what is the author's—or your– interpretation of these facts.)*

Historical *(what happened already)* **vs. predictive** *(what the author—or you—expect to happen in the future under these conditions)*

Open *(calls for an explanatory or descriptive answer; doesn't reveal the answer in the question)* **vs. closed** *(calls for a specific true answer, often answerable yes or no)*

Context-dependent *(the question is based on the specific details of the current situation)* **vs. context-free** *(the question is usable in a wide range of situations—it asks about the situation but was written independently of it).*



Gause / Weinberg is a superb source for context-free questions

More questions

Causal *(Why did this happen? Why is the author saying that?)*

Ask for evidence *(What proof is provided? Why should you believe this?)*

Evidentiary sufficiency *(Is this conclusion adequately justified by these data?)*

Trustworthiness of the data *(Were the data collection and analysis methods valid and reliable?)*

Critical awareness *(What are the author's assumptions? What are your assumptions in interpreting this?)*

Clarification *(What does this mean? Is it restated elsewhere in a clearer way?)*

Comparison *(How is this similar to that?) and* **Contrast** *(How is this different from that?)*

Implications *(If X is true, does that mean that Y must also be true?)*

Affective *(How does the author (or you) feel about that?)*

Relational *(How does this concept, theme or idea relate to that one?)*

Problem-solving *(How does this solve that problem, or help you solve it?*

More questions

Relevance *(Why is this here? What place does it have in the message or package of information the author is trying to convey? If it is not obviously relevant, is it a distractor?)*

Author's comprehension *(Does the author understand this? Is the author writing in a way that suggests s/he is inventing a concept without having researched it?)*

Author credibility *(What basis do you have for believing the author knows what s/he is talking about?)*

Author perspective / bias *(What point of view is the author writing from? What benefit could the author gain from persuading you that X is true or desirable (or false, etc.)?)*

The Michigan Educational Assessment Association has some useful material at
http://www.meap.org/html/TT_QuestionTypes.htm

More questions

Application *(How can you apply what the author is saying? How does the author apply it?)*

Analysis *(Can you (does the author) break down an argument or concept into smaller pieces?)*

Synthesis *(Does the author (or can you) bring together several facts, ideas, concepts into a coherent larger concept or a pattern?)*

(More along these lines come from Bloom's taxonomy…)

The classic context-free questions

Traditional news reporters' questions:

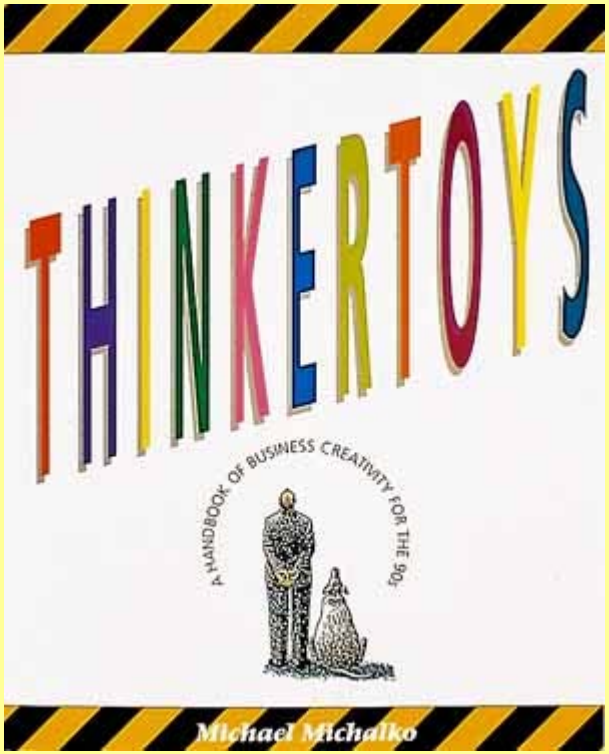
- **Who?**
- **What?**
- **When?**
- **Where?**
- **How?**
- **Why?**

For example, *Who will use this feature? What does this user want to do with it? Who else will use it? Why? Who will choose not to use it? What do they lose? What else does this user want to do in conjunction with this feature? Who is not allowed to use this product or feature, why, and what security is in place to prevent them?*

We use these in conjunction with questions that come out of the testing model (see below). The model gives us a starting place. We expand it by asking each of these questions as a follow-up to the initial question.

Using context-free questions to define a problem

- Why is it necessary to solve the problem?
- What benefits will you receive by solving the problem?
- What is the unknown?
- What is it that you don't yet understand?
- What is the information that you have?



- What is the source of this problem? (Specs? Field experience? An individual stakeholder's preference?)
- Who are the stakeholders?
- How does it relate to which stakeholders?
- What isn't the problem?
- Is the information sufficient? Or is it insufficient? Or redundant? Or contradictory?
- Should you draw a diagram of the problem? A figure?

Based on: *The CIA's Phoenix Checklists (Thinkertoys, p. 140) and Bach's Evaluation Strategies (Rapid Testing Course notes)*

Using context-free questions to define a problem

- Where are the boundaries of the problem?
- What product elements does it apply to?
- How does this problem relate to the quality criteria?
- Can you separate the various parts of the problem? Can you write them down? What are the relationships of the parts of the problem?
- What are the constants (things that can't be changed) of the problem?
- What are your critical assumptions about this problem?
- Have you seen this problem before?
- Have you seen this problem in a slightly different form?
- Do you know a related problem?
- Think of a familiar problem having the same or a similar unknown.
- Suppose you find a problem related to yours that has already been solved. Can you use it? Can you use its method?
- Can you restate your problem? How many different ways can you restate it? More general? More specific? Can the rules be changed?
- What are the best, worst, and most probable cases you can imagine?

Using context-free questions to evaluate a plan

- Will this solve the whole problem? Part of the problem?
- What would you like the resolution to be? Can you picture it?
- How much of the unknown can you determine?
- What reference data are you using (if any)?
- What product output will you evaluate?
- How will you do the evaluation?
- Can you derive something useful from the information you have?
- Have you used all the information?
- Have you taken into account all essential notions in the problem?
- Can you separate the steps in the problem-solving process? Can you determine the correctness of each step?
- What creative thinking techniques can you use to generate ideas? How many different techniques?
- Can you see the result? How many different kinds of results can you see?
- How many different ways have you tried to solve the problem?

Based on: *The CIA's Phoenix Checklists (Thinkertoys, p. 140) and Bach's Evaluation Strategies (Rapid Testing Course notes)*

Using context-free questions to evaluate a plan

- What have others done?
- Can you intuit the solution? Can you check the results?
- What should be done?
- How should it be done?
- Where should it be done?
- When should it be done?
- Who should do it?
- What do you need to do at this time?
- Who will be responsible for what?
- Can you use this problem to solve some other problem?
- What is the unique set of qualities that makes this problem what it is and none other?
- What milestones can best mark your progress?
- How will you know when you are successful?
- How conclusive and specific is your answer?

Context-Free Questions

Context-free process questions

- Who is the client?
- What is a successful solution worth to this client?
- What is the real (underlying) reason for wanting to solve this problem?
- Who can help solve the problem?
- How much time is available to solve the problem?

Context-free product questions

- What problems could this product create?
- What kind of precision is required / desired for this product?

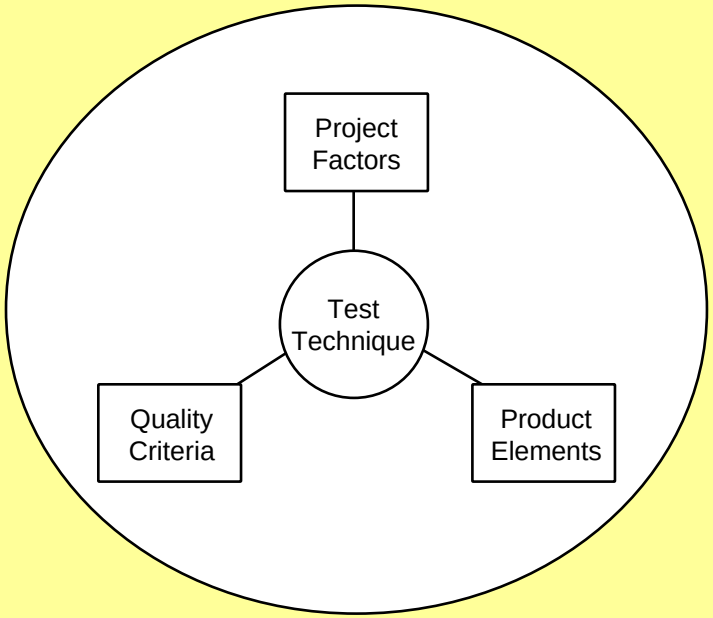
Metaquestions (when interviewing someone for info)

- Am I asking too many questions?
- Do my questions seem relevant?
- Are you the right person to answer these questions?
- Is there anyone else who can provide additional information?
- Is there anything else I should be asking?
- Is there anything you want to ask me?
- May I return to you with more questions later?

A sample of additional questions based on Gause & Weinberg's **Exploring Requirements p. 59-64**

An active reading example

To find and organize the claims, I use an active reading approach based on the Heuristic Test Strategy Model



We'll do this in our next section of the tutorial

Spec testing issues

What is the specification?

What does the specification say?

Critiquing the specification (what it says):

- How it says what it says
- What it says about the product
- What it says about the testing of the product

Critiquing the specification (doing the critique)

Driving tests from the specification

Legal issues

How it says what it says

Ambiguity

- Are multiple interpretations possible? Likely?

Adequacy

- Does it provide enough information for programming, documentation and testing?

Completeness

- To what extent does it cover the
  - Feature set
  - Use cases
  - Usage scenarios
  - Test-relevant information (such as boundaries, error handling, etc.)

Ambiguity analysis

Many sources of ambiguity in software design & development.

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

Whenever there is ambiguity, there is a strong opportunity for a defect

- Richard Bender teaches this well in his courses on Requirements Based Testing. His course has some great labs, and he coaches well. I recommend it. If you can't take his course, you can find notes based on his work in Rodney Wilson's Software RX: Secrets of Engineering Quality Software.
- An interesting workbook: Cecile Spector, Saying One Thing, Meaning Another. She discusses and provides examples and exercises with many additional ambiguities in common English than I can cover here.

Common ambiguities in use of the language

Undefined words

- "The user may authenticate incoming documents by processing their security attributes."

Incorrectly used words

- *Typeface* refers to a set of characters having the same design, or to the design. *Font* refers to a specific size and style of a typeface. (See google: *define typeface* and *define font*.) A version of OpenOffice labeled a list of typefaces as fonts and a list of styles (italics, bold, etc.) as typefaces. How would you interpret help documentation that referred to "typefaces" ?

Contradictorily defined words

- Use "valid" to mean (sometimes) a value considered valid by a user and (other times) a value that meets input criteria constraints in a program.

Vague words

- Etc., will display a message, process, upgrade, performance, user friendly

Commonly misunderstood words

- *i.e.* (means *id est* = *that is* and calls for a restatement or redefinition of a previous word or statement) whereas *e.g.* means *exampli gratia* (for example)

Ambiguous quantities

- Within, between, up to, almost, on the order of

Impossible promises

- "The program will be fully tested." "Performance will be instantaneous."

Common ambiguities: Logical conditions

Incomplete set of logical conditions

- If A and B then C. If A and not B then D
  - What about B and not A?

Logical operators ambiguously grouped

- If A and B or C then D
  - Is this (A and B) or C? Is it A and (B or C)?
  - Just because precedence orders are defined by convention doesn't mean that the spec author, the spec reviewers, and the programmers know them

Negation without explicit specification of scope

- If not A and B then D
  - Is this (Not A) and B? Is it Not (A and B)? Is it Not-A and Not-B?

There are plenty more of these. Look at any logic text.

Common ambiguities: Missing facts (1)

Unspecified decision maker

- If X is unacceptable, then
  - Unacceptable according to who?

Assumes facts not specified

- Spec assumes the reader is familiar with the specifics of regulations, environmental constraints, etc. These might change or differ across countries, platforms, etc.

Ambiguity in time

- Does X have to precede Y? In the statement, "Do A if X happens and Y happens and Z happens" does it matter if they happen in that order?

Causes without effects

- The case X is greater than Y will trigger special processing

Effects without causes

- If X occurs during processing, then …

Effects with underspecified causes

- General protection fault

Common ambiguities: Missing facts (2)

Unspecified error handling

- "The program will accept up to 3 names."

Unspecified variables

- The program will set a flag if this happens.
  - What flag?

Boundaries unspecified or underspecified

- Is 0 a positive number? If 0<x<100 is valid, how big is the maximum value that you will allow to be copied into X for evaluation?
  - (Whittaker's testing approach rests on programmers being blind to a wide range of unspecified system or program constraints)

Unspecified quantities

- The program will compare the value input for X to the maximum allowed

Mentioned but undefined cases

- "The page format dialog will display 3 column width fields at a time. The user may not specify more than 10 columns."

Ambiguity analysis: Break statements into elements

Gause & Weinberg

- "Mary had a little lamb" (read the statement several times, emphasizing a different word each time and asking what the statement means, read that way)
- "Mary conned the trader" (for each word in the statement, substitute a wide range of synonyms and review the statement's resulting meaning.)

"Slice & dice" (Thinkertoys)

- Make / read a statement about the program. Work through the statement one word at a time, asking what each word means or implies.

These approaches can help you ferret out ambiguity in the product definition. By seeing how different people can interpret a key statement in the spec, you can imagine new tests to check which meaning is operative in the program.

Break statements into elements:

Quality is value to some person

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

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

What it says about the product

Correctness

- Does it accurately describe the program?

Controversy

- Which parts are controversial? Who are the stakeholders who disagree and why do they disagree?

Adequacy

- Does it provide enough information for programming, documentation and testing?

Completeness

- Does it cover the feature set?

Design

- Can you tell whether it specifies design errors?
- Is it understandable, usable, trainable, consistent, appropriate for the market?
- Does it set up the program / programmer for common errors?

What it says about testing

Early in the project, you can review the spec's implications for testing, and change them or prepare for them.

- Implications for test design
  - What test techniques will be most appropriate for this project?
  - Will you need additional training or tools for them?
  - Are there ways to simplify (or otherwise change) to product in ways that would call for simpler or cheaper or more easily structured techniques?
  - How much exploring will this project require?
    - > Does your staff have the knowledge, skills and connections?
- Test schedule and resource commitments / implications
  - When will you receive deliverables from others?
  - When are you to deliver your work?
  - What do you need to get this done?
  - Are any of your commitments unreasonable?
- Testability support

Design reviews: Testability

Controllability

Observability

Availability

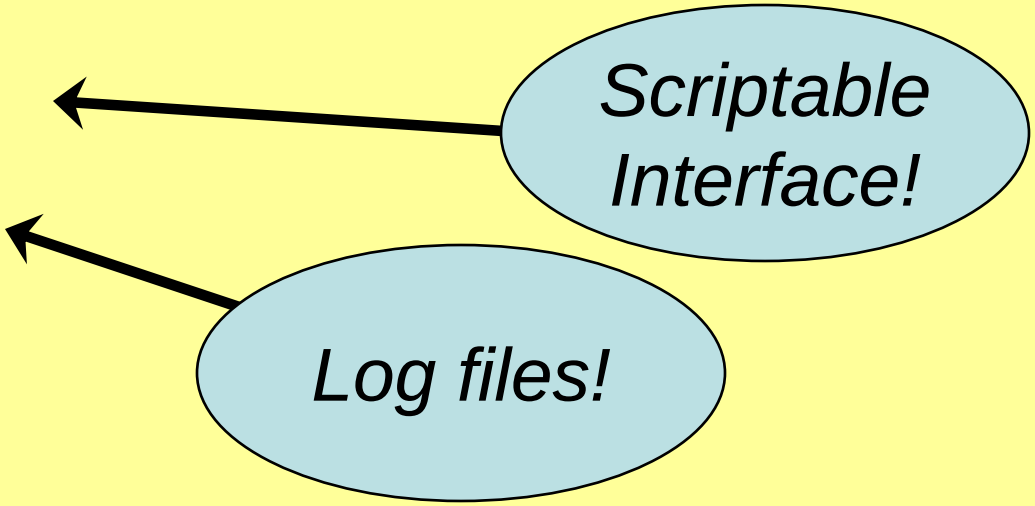
Simplicity

Stability

Information

Separation of functional components

Availability of oracles



Testing is far more rapid when the product is far more testable

Testing the program against the spec

What is the specification?

What does the specification say?

Critiquing the specification (what it says):

- How it says what it says
- What it says about the product
- What it says about the testing of the product

Critiquing the specification (doing the critique)

Driving tests from the specification

Legal issues

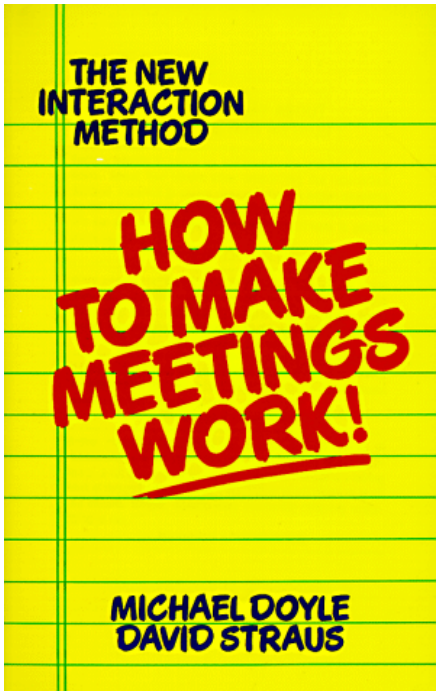
Critiquing specs: Process notes

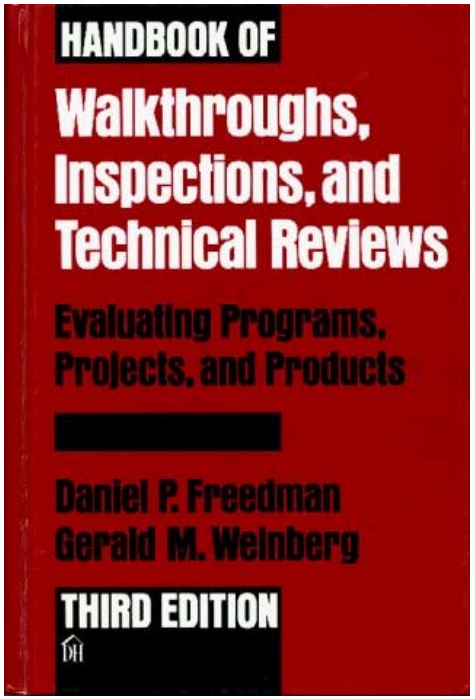
Review meetings

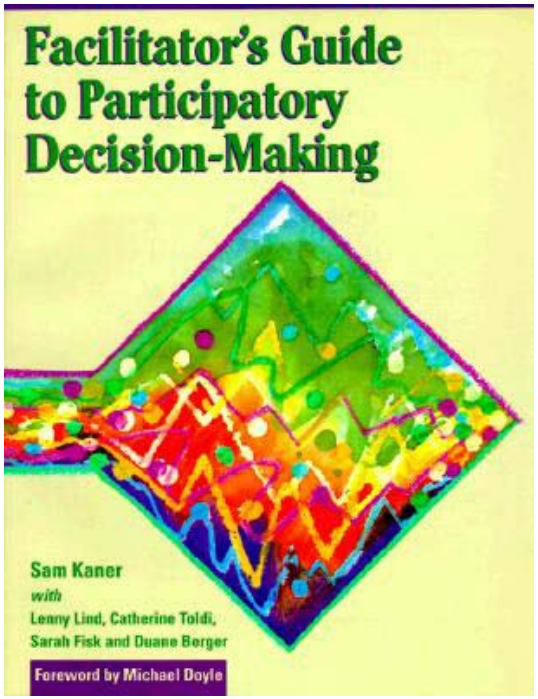
- Test groups often train to facilitate technical reviews

Detailed comments on the specification

- Same guidelines as for critiquing other tech pubs. See *Testing Computer Software*







Spec testing issues

What is the specification?

What does the specification say?

Critiquing the specification (what it says):

- How it says what it says
- What it says about the product
- What it says about the testing of the product

Critiquing the specification (doing the critique)

Driving tests from the specification

Legal issues

Driving tests from the specification

Who are the stakeholders?

- There are stakeholders for all services. Who are yours?
  - Regulators? Marketing? End customer?
  - Journalists? Attorney? Court? (Expert witness?)
  - Client company (you're the outsource test lab)?
- These stakeholders would have different test-result / test-documentation expectations from the typical project team.

What is a good specification driven test?

- Same as "what is a good test?"
- But tests come from specs
- Might be that a test that covers several spec items is preferred to a single-item test
- Might be that tests that resolve or expose and show implications of specification ambiguities are particularly important

Driving tests from the specification

Coverage

- Key issue is coverage of the specification
  - Cover items (individual statements)
    - > But how many tests per statement do you need?
    - > Many groups require only one per spec assertion
  - Cover specified relationships
    - > To test `A && B`
    - > You probably want to test at least `A true and B true`
    - > `A true and B false`
    - > `A false and B true`

Brian Marick's *multi* tool is useful for this

Students at Florida Tech are now publishing a Release 2.0 of multi (see www.testingeducation.org in December

Driving tests from the spec: Coverage

Important to understand the level of generality called for when testing a spec item. For example, imagine a field X:

- We could test a single use of X
- Or we could partition possible values of X and test boundary values
- Or we could test X in various scenarios
- Which is the right one?
- This partially depends on whether specification-driven testing is your exclusive style of testing

How do we track coverage?

- Trace tests BACK TO the specification with traceability matrices

Traceability matrix

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

Traceability matrix

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

The rows are test cases.

The cells show which test case tests which items.

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

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

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

Traceability tool risk—test case management tools can drive you into wasteful over-documentation and unmaintainable repetition

Spec testing issues

What is the specification?

What does the specification say?

Critiquing the specification (what it says):

- How it says what it says
- What it says about the product
- What it says about the testing of the product

Critiquing the specification (doing the critique)

Driving tests from the specification

Legal issues

Legal issues

Warranties based on claims to the public

- Article: *Liability for defective documentation*

  http://www.kaner.com/pdfs/liability_sigdoc.pdf

Warranties based on claims to custom-product customer

Claims of compatibility with other products

- Article: *Liability for product incompatibility*

  http://www.kaner.com/pdfs/liability_sigdoc.pdf

Errors in your product documents, that are not about your products

- Article: *Liability for defective content*

  http://www.kaner.com/pdfs/sigdocContent.pdf

Testing claims against the product

Uniform Commercial Code Article 2 (2003 revision)

SECTION 2-313A. (2) If a seller in a record packaged with or accompanying the goods makes an affirmation of fact or promise that relates to the goods, provides a description that relates to the goods, or makes a remedial promise, and the seller reasonably expects the record to be, and the record is, furnished to the remote purchaser, the seller has an obligation to the remote purchaser that:

(a) the goods will conform to the affirmation of fact, promise or description unless a reasonable person in the position of the remote purchaser would not believe that the affirmation of fact, promise or description created an obligation; and

(b) the seller will perform the remedial promise.

(3) It is not necessary to the creation of an obligation under this section that the seller use formal words such as "warrant" or "guarantee" or that the seller have a specific intention to undertake an obligation, but an affirmation merely of the value of the goods or a statement purporting to be merely the seller's opinion or commendation of the goods does not create an obligation.

Using the Satisfice Heuristic Test Strategy Model to guide analysis

Reviewing a document with the Heuristic Test Strategy Model

- The last section has many slides on active reading.
- In the last exercise, we reviewed the requirements document on its own terms.
  - We see what is there and come to understand it better.
- Active readers often operate from a different organizational structure, fitting the information from the document under review into the structure they are trying to fill rather than being bound by the structure of the document.
- We demonstrate what active reading is about in this exercise, by using an independently created structure (the Heuristic Test Strategy Model) as the base document and reviewing the specification in terms of how well we can map its information onto the information structure of HSTM.

Heuristic Test Strategy Model

Authored by James Bach

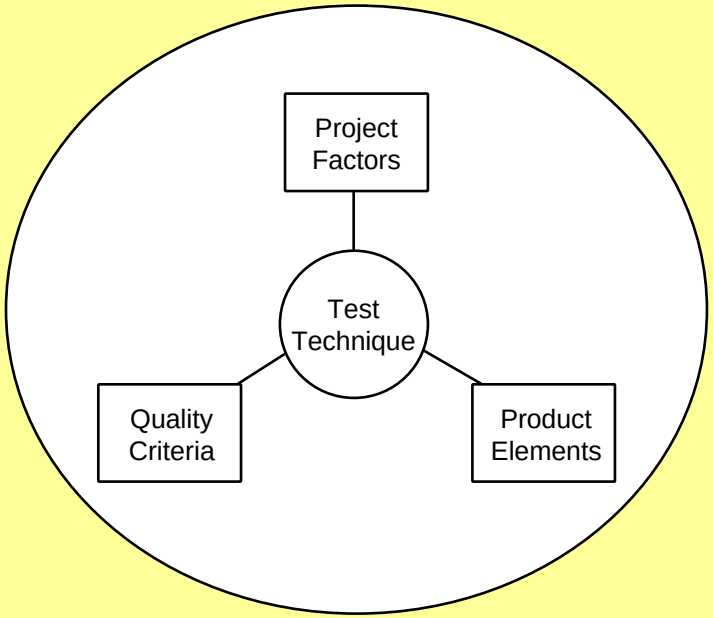
- 10 years of critical peer review by colleagues.
- Several of us have found this a very useful tool for
  - Guiding exploration (see Bach's and Bolton's courses)
  - Structuring a failure mode and effects analysis
    - > See Giri Vijayaraghavan & Cem Kaner Bug taxonomies: Use them to generate better tests at http://www.kaner.com/pdfs/BugTaxonomies.pdf and Giri's thesis, "A Taxonomy of E-Commerce Risks and Failures." at http://www.testingeducation.org/a/tecrf.pdf
    - > Another thesis on mobile wireless apps coming soon by Ajay Jha
  - Specification analysis (my primary use of the model)

An active reading example

To find and organize the claims, I use an active reading approach based on the Heuristic Test Strategy Model

As you read the spec,

- Start from the assumption that every sentence in the spec is meant to convey information.
- Take four writing pads, mark them *Project*, *Product*, *Quality* and *To-Do*.



- On the appropriate pad, note briefly what the spec tells you about:
  - the project and how it is structured, funded or timed, or
  - the product (what it is and how it works) or
  - the quality criteria you should evaluate the product against or
  - things you need to do, that you learned from the spec.

An active reading example

As you note what you *have* discovered, make additional notes in a different pen color, such as:

- Items that haven't yet been specified, that you think are relevant.
- References to later parts of the specification or to other documents that you'll need to understand the spec.
- Questions that come to mind about how the product works, how the project will be run or what quality criteria are in play.
- Your disagreements or concerns with the product / project as specified.

Beware of getting too detailed in this. If the spec provides a piece of information, you don't need to rewrite it. Just write down a pointer (and a spec page number). Your list is a quick summary that you build as you read, to help you read, not a rewriting of the document.

As you read further, some of your earlier questions will be answered. Others won't. Ask the programmers or spec writers about them.

Heuristic test strategy model

The HSTM is another example of a tool that is especially useful for auditing / mentoring purposes.

It provides you a support structure for discovering what is missing or buried in someone else's work.

We have seen this already in the ET Dynamics handout.

My bug appendix in Testing Computer Software was widely used for that, and HSTM has been the root of comparable, but more recent documents (e.g. Vijayaraghavan's thesis).

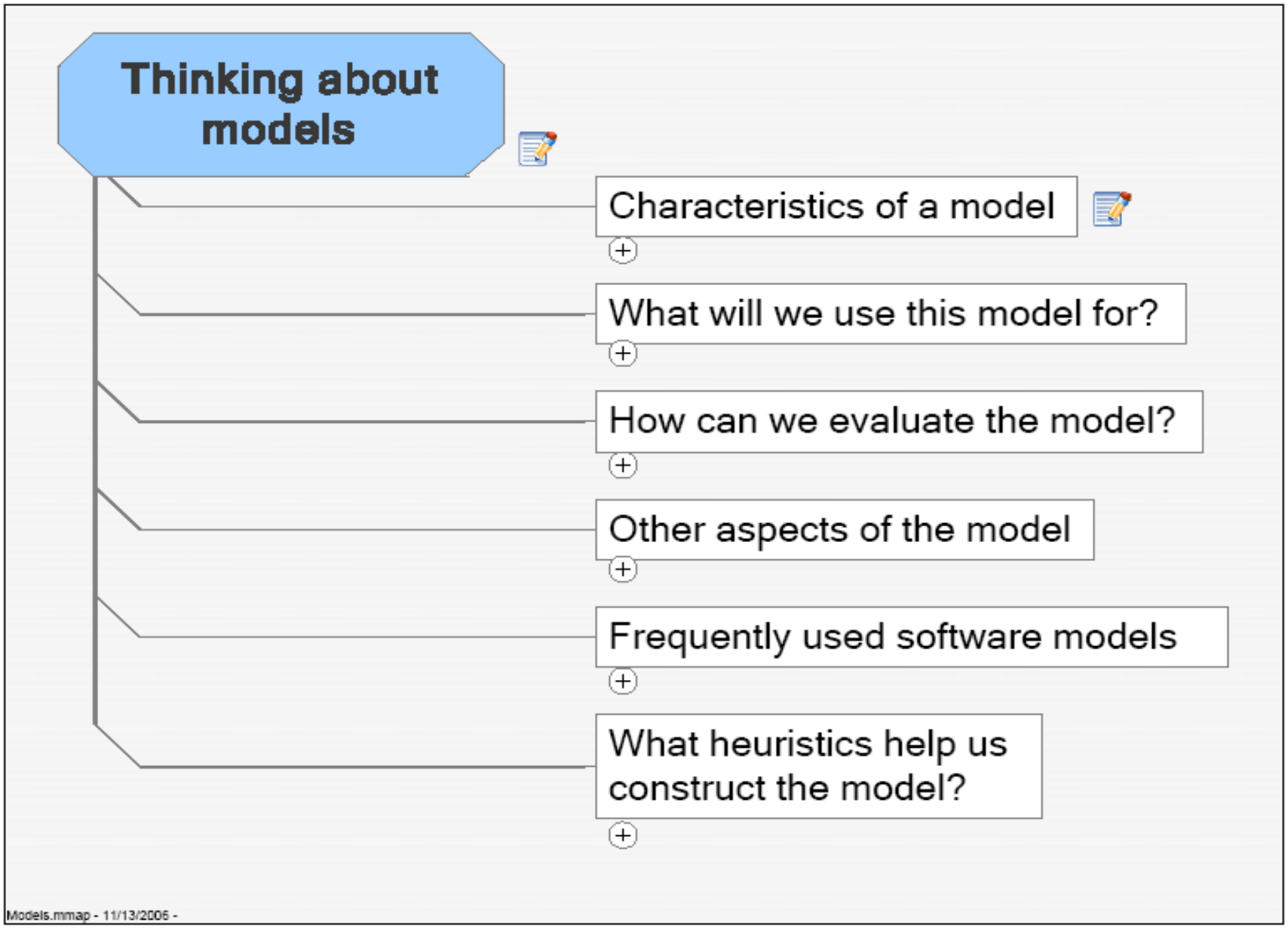
The Phoenix questions in the previous section provide another strong example of a question set that is at least as useful for post-creation review as for initial planning.

Using and developing models in software testing

Models

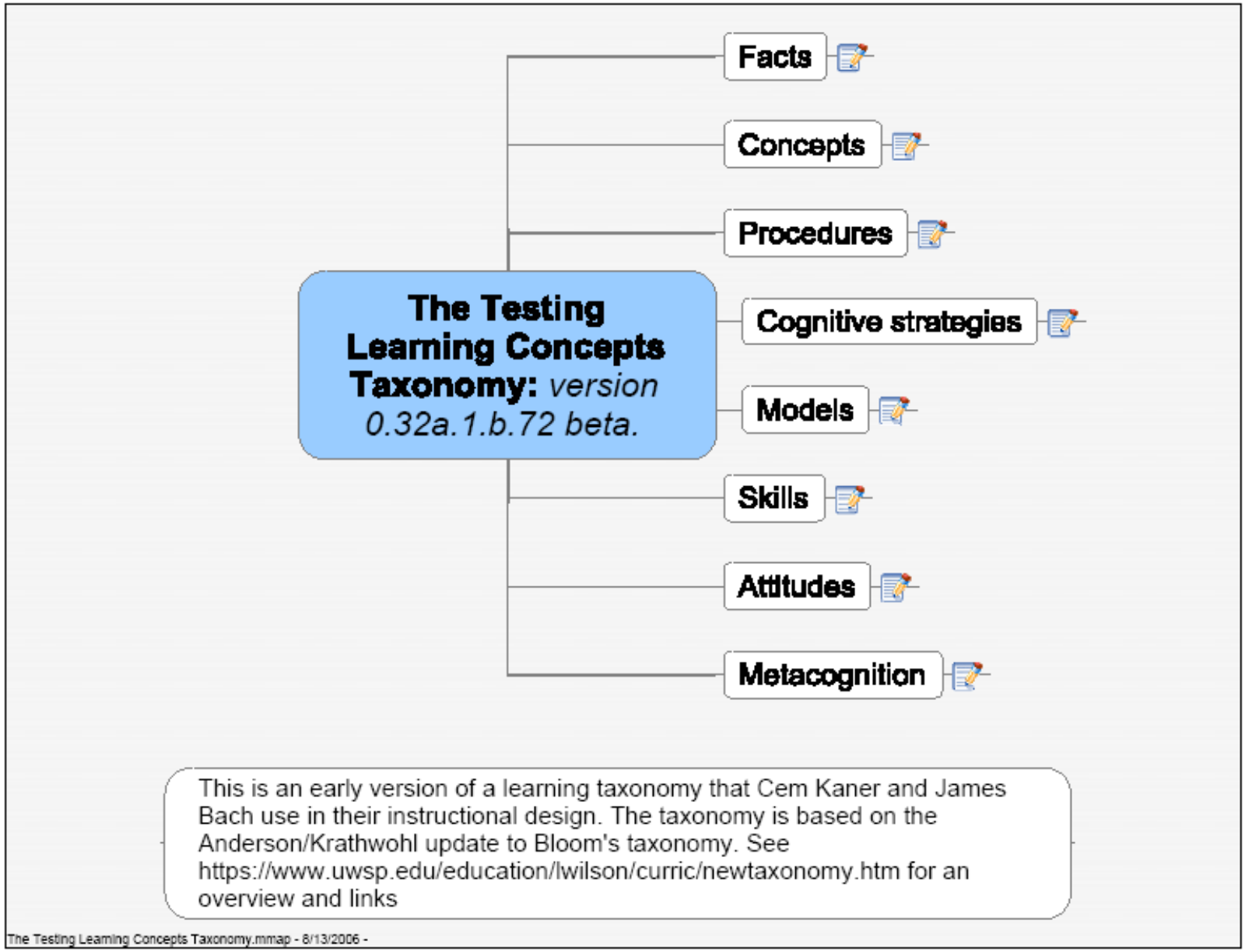
A model is a simplified formal representation of a relationship, process or system. The simplification makes some aspects of the thing modeled clearer, more visible, and easier to work with.

- All tests are based on models
- Many of our models are implicit
- When the behavior of a program "feels wrong," it is clashing with your internal model of the program (and how it should behave)


Thinking about models
Characteristics of a model
What will we use this model for?
How can we evaluate the model?
Other aspects of the model
Frequently used software models
What heuristics help us construct the model?
Models.mmap - 11/13/2006 -

A taxonomy model (instructional design / assessment)

	COGNITIVE PROCESSES					
KNOWLEDGE DIMENSIONS	Remember	Understand	Apply	Analyze	Evaluate	Create
Facts						
Concepts						
Procedures						
Cognitive strategies						
Models						
Skills						
Attitudes						
Metacognition						


Facts
Concepts
Procedures
The Testing Learning Concepts Taxonomy: version 0.32a.1.b.72 beta.
Cognitive strategies
Models
Skills
Attitudes
Metacognition
This is an early version of a learning taxonomy that Cem Kaner and James Bach use in their instructional design. The taxonomy is based on the Anderson/Krathwohl update to Bloom's taxonomy. See https://www.uwsp.edu/education/lwilson/curric/newtaxonomy.htm for an overview and links
The Testing Learning Concepts Taxonomy.mmap - 8/13/2006 -

From information to test:

Using quicktests

QuickTests?

A ***quicktest*** is a cheap test that has some value but requires little preparation, knowledge, or time to perform.

- Participants at the 7th Los Altos Workshop on Software Testing (Exploratory Testing, 1999) pulled together a collection of these.
- James Whittaker published another collection in *How to Break Software*.
- Elisabeth Hendrickson teaches courses on bug hunting techniques and tools, many of which are quicktests or tools that support them.

A Classic QuickTest: The Shoe Test

Find an input field, move the cursor to it, put your shoe on the keyboard, and go to lunch.

Basically, you're using the auto-repeat on the keyboard for a cheap stress test.

- *Tests like this often overflow input buffers.*

In Bach's favorite variant, he finds a dialog box so constructed that pressing a key leads to, say, another dialog box (perhaps an error message) that also has a button connected to the same key that returns to the first dialog box.

- *This will expose some types of long-sequence errors (stack overflows, memory leaks, etc.)*

Another Classic Example of a QuickTest

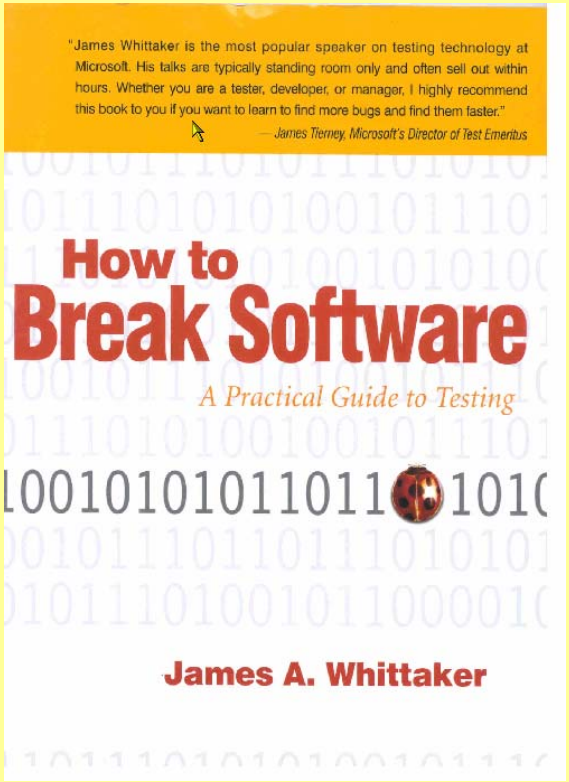
Traditional boundary testing

- All you need is the variable, and its possible values.
- You need very little information about the meaning of the variable (why people assign values to it, what it interacts with).
- You test at boundaries because miscoding of boundaries is a common error.

Note the foundation of this test.

> There is a programming error so common that it's worth building a test technique optimized to find errors of that type.

"Attacks" to expose common coding errors



Jorgensen & Whittaker pulled together a collection of common coding errors, many of them involving insufficiently or incorrectly constrained variables.

They created (or identified common) *attacks* to test for these.

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

Think back to boundary testing:

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

"Attacks" to expose common coding errors

In his book, How to Break Software, Professor Whittaker expanded the list and, for each attack, discussed

- When to apply it
- What software errors make the attack successful
- How to determine if the attack exposed a failure
- How to conduct the attack, and
- An example of the attack.

We'll list How to Break Software's attacks here, but recommend the book's full discussion.

"Attacks" to expose common coding errors

User interface attacks: Exploring the input domain

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

~From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

User interface attacks: Exploring outputs

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

~From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

Testing from the user interface: Data and computation

Exploring stored data

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

~From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

Testing from the user interface: Data and computation

Exploring computation and feature interaction

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

~From Whittaker, *How to Break Software*

"Attacks" to expose common coding errors

System interface attacks

Testing from the file system interface: Media-based attacks

- Attack 1: Fill the file system to its capacity
- Attack 2: Force the media to be busy or unavailable
- Attack 3: Damage the media

Testing from the file system interface: File-based attacks

- Attack 4: Assign an invalid file name
- Attack 5: Vary file access permissions
- Attack 6: Vary or corrupt file contents

~From Whittaker, *How to Break Software*

Additional QuickTests from LAWST

Several of the tests we listed at LAWST (7th Los Altos Workshop on Software Testing, 1999) are equivalent to the attacks later published by Whittaker.

He develops the attacks well, and we recommend his descriptions.

LAWST generated several other quicktests, including some that aren't directly tied to a simple fault model.

Many of the ideas in these notes were reviewed and extended by the other LAWST 7 attendees: Brian Lawrence, III, Jack Falk, Drew Pritsker, Jim Bampos, Bob Johnson, Doug Hoffman, Chris Agruss, Dave Gelperin, Melora Svoboda, Jeff Payne, James Tierney, Hung Nguyen, Harry Robinson, Elisabeth Hendrickson, Noel Nyman, Bret Pettichord, & Rodney Wilson. We appreciate their contributions.

Additional QuickTests

Interference testing

We look at asynchronous events here. One task is underway, and we do something to interfere with it.

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

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

Additional QuickTests

Interference testing: Generate interrupts

- **from a device related to the task**
  - e.g. pull out a paper tray, perhaps one that isn't in use while the printer is printing
- **from a device unrelated to the task**
  - e.g. move the mouse and click while the printer is printing
- **from a software event**
  - e.g. set another program's (or this program's) time-reminder to go off during the task under test

Additional QuickTests

Interference: Change something this task depends on

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

Additional QuickTests

Interference testing: Cancel

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

Additional QuickTests

Interference testing: Pause

Find some way to create a temporary interruption in the task.

- Pause the task
  - for a short time
  - for a long time (long enough for a timeout, if one will arise)
- For example,
  - Put the printer on local
  - Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

Additional QuickTests

Interference testing: Swap (out of memory)

Swap the process out of memory while it's running

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

Swap a related process out of memory while the process under test is running.

Additional QuickTests

Interference testing: Compete

Examples:

Compete for a device (such as a printer)

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

Compete for processor attention

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

Send this process another job while one is underway

Additional QuickTests

Follow up recent changes

Code changes cause side effects

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

Additional QuickTests

Explore data relationships

Pick a data item

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

Additional QuickTests

Explore data relationships

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

We discuss this table more in our lectures on combination testing.

Additional QuickTests

Explore data relationships (continued)

Many possible relationships. For example,

- V1 < V2+K (V1 is constrained by V2+K)
- V1 = f(V2), where f is any function
- V1 is an enumerated variable but the set of choices for V1 is determined by the value of V2

Relations are often reciprocal, so if V2 constrains V1, then V1 might constrain V2 (try to change V2 after setting V1)

Given the relationship,

- Try to enter relationship-breaking values everywhere that you can enter V1 and V2.
- Pay attention to unusual entry options, such as editing in a display field, import, revision using a different component or program

Once you achieve a mismatch between V1 and V2, the program's data no longer obey rules the programmer expected would be obeyed, so anything that assumes the rules hold is vulnerable. Do follow-up testing to discover serious side effects of the mismatch

Even More QuickTests (from Bach's Rapid Testing Course)

Quick tours of the program

Variability Tour: Tour a product looking for anything that is variable and vary it. Vary it as far as possible, in every dimension possible.

- ***Exploring variations is part of the basic structure of Bach's testing when he first encounters a product.***

Complexity Tour: Tour a product looking for the most complex features and data. Create complex files.

Sample Data Tour: Employ any sample data you can, and all that you can. The more complex the better

Even More QuickTests (from Bach's Rapid Testing Course)

Continuous Use: While testing, do not reset the system. Leave windows and files open. Let disk and memory usage mount. You're hoping the system ties itself in knots over time.

Adjustments: Set some parameter to a certain value, then, at any later time, reset that value to something else without resetting or recreating the containing document or data structure.

Dog Piling: Get more processes going at once; more states existing concurrently. Nested dialog boxes and non-modal dialogs provide opportunities to do this.

Undermining: Start using a function when the system is in an appropriate state, then change the state part way through (for instance, delete a file while it is being edited, eject a disk, pull net cables or power cords) to an inappropriate state. This is similar to interruption, except you are expecting the function to interrupt itself by detecting that it no longer can proceed safely.

Even More QuickTests (from Bach's Rapid Testing Course)

Error Message Hangover: Make error messages happen. Test hard after they are dismissed. Developers often handle errors poorly. Bach once broke into a public kiosk by right clicking rapidly after an error occurred. It turned out the security code left a 1/5 second window of opportunity for me to access a special menu and take over the system.

Click Frenzy: Testing is more than "banging on the keyboard", but that phrase wasn't coined for nothing. Try banging on the keyboard. Try clicking everywhere. Bach broke into a touchscreen system once by poking every square centimeter of every screen until he found a secret button.

Multiple Instances: Run a lot of instances of the application at the same time. Open the same files.

Feature Interactions: Discover where individual functions interact or share data. Look for any interdependencies. Tour them. Stress them. Bach once crashed an app by loading up all the fields in a form to their maximums and then traversing to the report generator.

Even More QuickTests (from Bach's Rapid Testing Course)

Cheap Tools! Learn how to use InCtrl5, Filemon, Regmon, AppVerifier, Perfmon, Task Manager (all of which are free). Have these tools on a thumb drive and carry it around. Also, carry a digital camera. Bach carries a tiny 3 megapixel camera and a tiny video camera in his coat pockets. He uses them to record screen shots and product behaviors.

- Elisabeth Hendrickson suggests several additional tools at http://www.bughunting.com/bugtools.html

Resource Starvation: Progressively lower memory and other resources until the product gracefully degrades or ungracefully collapses.

Play "Writer Sez": Look in the online help or user manual and find instructions about how to perform some interesting activity. Do those actions. Then improvise from them. Often writers are hurried as they write down steps, or the software changes after they write the manual.

Even More QuickTests (from Bach's Rapid Testing Course)

Crazy Configs: Modify O/S configuration in non-standard or non-default ways either before or after installing the product. Turn on "high contrast" accessibility mode, or change the localization defaults. Change the letter of the system hard drive.

Grokking: Find some aspect of the product that produces huge amounts of data or does some operation very quickly. For instance, look a long log file or browse database records very quickly. Let the data go by too quickly to see in detail, but notice trends in length or look or shape of the patterns as you see them.

Parlour Tricks are not Risk-Free

These tricks can generate lots of flash in a hurry

- The DOS disk I/O example
- The Amiga clicky-click-click-click example

As political weapons, they are double-edged

- If people realize what you're doing, you lose credibility
- Anyone you humiliate becomes a potential enemy

Some people (incorrectly) characterize exploratory testing as if it were a collection of quicktests.

As test design tools, they are like good candy

- Yummy
- Everyone likes them
- Not necessarily nutritious. (You may never get to the deeper issues of the program.)

The challenge of relevance

The relevance problem as a test design problem

- We often go from technique to test
  - Find all variables, domain test each
  - Find all spec paragraphs, make a relevant test for each
  - Find all lines of code, make a set of tests that collectively includes each
- It is much harder to go from a risk to a test
  - The program will crash?
  - The program will have a wild pointer?
  - The program will have a memory leak?
  - The program will be hard to use?
  - The program will corrupt its database?

The relevance problem as a test design problem

- The challenge of exploratory testing is often to take a test idea (especially potential problem)
  - maybe learned from study of competitor's product, or support history, or failure of other products on this operating system or written in this programming language
- And turn the test idea into one or more tests

How do we map from a test idea to a test?

How do we map from a test idea to a test?

- I don't have a general answer.
- Cross-mapping of knowledge is one of (perhaps *the*) most difficult cognitive tasks.
  - Ability to do this is often discussed in terms of "G" ("general intelligence", the hypothetical dominant factor that underlies IQ scores)

How do we map from a test idea to a test?

- When it is not clear how to work backwards to the relevant test, four tactics sometimes help:
  - Ask someone for help
  - Ask google for help. (Look for discussions of the type of failure; look for discussions of different faults and see what types of failures they yield)
  - Review your toolkit of techniques, searching for a test type with relevant characteristics
  - Turn the failure into a story and gradually evolve the story into something you can test from
- There are no guarantees in this, but you get better at it as you practice, and as you build a broader inventory of techniques.

An overview

of test techniques

How do we use test techniques to create tests?

Analyze the situation.

Model the test space.

Select what to cover.

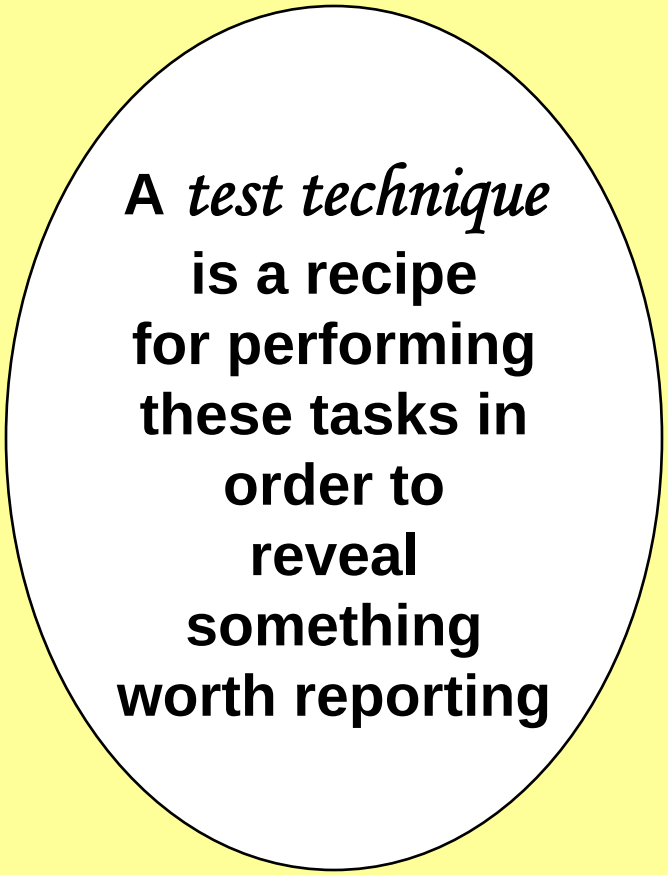
Determine test oracles.

Configure the test system.

Operate the test system.

Observe the test system.

Evaluate the test results.



Designing test cases

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,

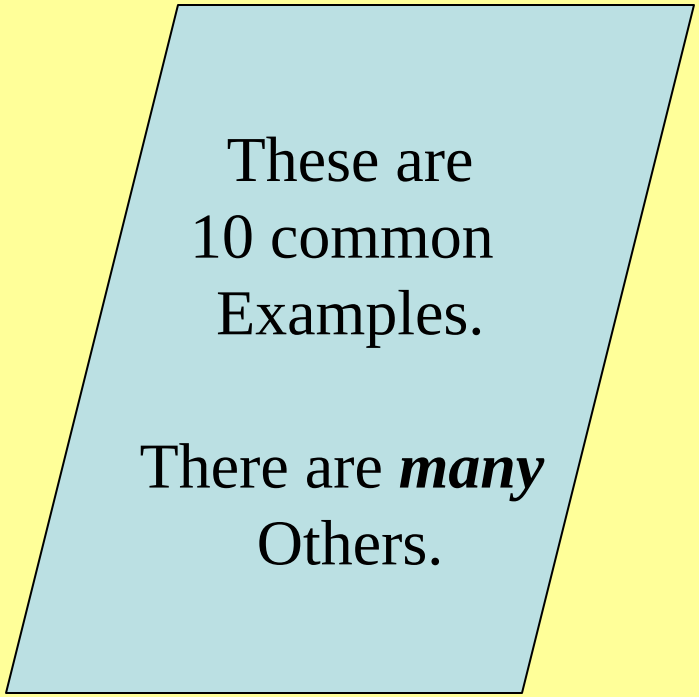
> A test case is a question you ask the program.

The point of running the test is to gain information, for example whether the program will pass or fail the test.

- 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.

Ten dominating techniques

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



Ten dominating techniques

- FUNCTION TESTING
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- State-model based testing
- High volume automated testing

Test each feature or function on its own.

Scan through the product, covering every feature or function with at least enough testing to determine what it does and whether it is working.

Ten dominating techniques

- Function testing
- SPECIFICATION-BASED TESTING
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- State-model based testing
- High volume automated testing

Check every claim made in the reference document (such as, a contract specification).

Test to the extent that you have proved the claim true or false.

Ten dominating techniques

- Function testing
- Specification-based testing
- DOMAIN TESTING
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- State-model based testing
- High volume automated testing

Focus on variables, such as inputs, outputs, configuration, or internal (e.g. file-handling) variables.

For every variable or combination of variables, consider the space of possible values. Simplify it by partitioning into subsets. Pick a few representatives of each subset.

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- RISK-BASED TESTING
- Scenario testing
- Regression testing
- Stress testing
- User testing
- State-model based testing
- High volume automated testing

A program is a collection of opportunities for things to go wrong.

For each way that you can imagine the program failing, design tests to determine whether the program actually will fail in that way.

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- SCENARIO TESTING
- Regression testing
- Stress testing
- User testing
- State-model based testing
- High volume automated testing

Tests are complex stories that capture how the program will be used in real-life situations.

These are combination tests, whose combinations are credible reflections of real use.

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- REGRESSION TESTING
- Stress testing
- User testing
- State-model based testing
- High volume automated testing

Repeat the same test after some change to the program.

You can use any test as a regression test, but if you do a lot of regression testing, you will (or should) learn to design cases for efficient reuse.

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- STRESS TESTING
- User testing
- State-model based testing
- High volume automated testing

Many definitions of stress testing.

When I say stress testing, I mean tests intended to overwhelm the product, to subject it to so much input, so little memory, such odd combinations that I expect it to fail and am exploring its behavior as (and after) it fails.

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- USER TESTING
- State-model based testing
- High volume automated testing

Give the program to "a user," see what he does with it and how it responds.

User tests can be tightly structured or very loosely defined. The essence is room for action and response by "users".

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- STATE-MODEL BASED TESTING
- High volume automated testing

Model the program as a state machine that runs from state to state in response to events (such as new inputs).

In each state, does it respond correctly to each event?

Ten dominating techniques

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- State-model based testing
- HIGH VOLUME AUTOMATED TESTING

Program the computer to design, implement, execute and interpret a large series of tests.

You set the wheel in motion, supply the oracle(s) and evaluate the pattern of results.

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)*.

Non-redundant. This test represents a larger group that address the same risk.

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 prevents you from doing other work

Differences in the emphasis on the goodness-of-test attributes are the key differences between test techniques

Domain testing

- Focused on non-redundancy, validity, power, and variables-coverage. Tests are typically highly repeatable, simple, and *should be* easy to maintain.
- *Not* focused on representativeness, credibility, or motivational effect.

Scenario testing

- Focused on validity, complexity, credibility, and motivational effect.
- *Not* focused on power, maintainability, or coverage.

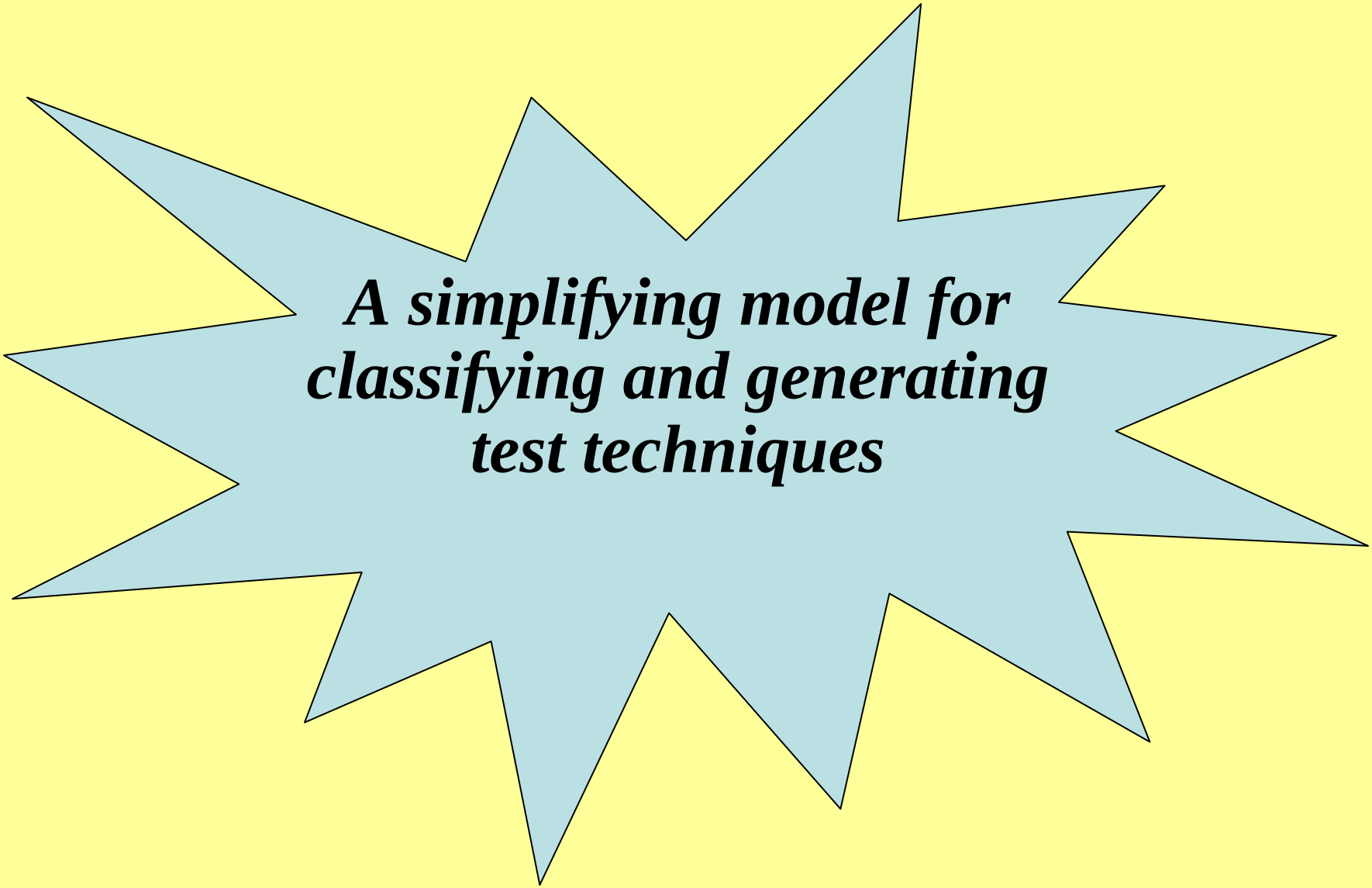
"Not focused" doesn't mean, "never is." It means that this is a factor that we don't treat as critical in developing or evaluating this type of test.

How to choose a test technique?

This is a multi-dimensional challenge:

- Your information objectives
- Attributes of the potential tests
- ***The development context:***
  - ***Product elements***
  - ***Quality criteria***
  - ***Risks***
  - ***Project factors (constraints and opportunities)***

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.***
- **Artefact**: ***What you will report***

All testing involves all five dimensions. Individual techniques focus on 1 or 2 dimensions, leaving the others float free

Example of technique emphasis

What is the difference between

- User testing?
- Usability testing?
- User interface testing?

Getting back to relevance

If you don't have a technique at hand, you will often have to invent your own.

Or at least, polish a test idea into a good test.

This is especially true with stories that give an initial approach to a risk but need work.

Example:

> Joe bought a smart refrigerator that tracks items stored in the fridge and prints out grocery shopping lists. One day, Joe asked for a shopping list for his usual meals in their usual quantities and the fridge crashed with an unintelligible error message.

How would you test for this bug?

Enhancing the test case from the story

- We start with Joe and his failure.
- We generate hypotheses for situations that *might* lead to a failure like that:
  - Wild pointer
  - Stack overflow
  - Unusual timing condition
  - Unusual collection of things in the fridge
- Now the trick is to refine the hypotheses into harsher and harsher tests
- Until we are satisfied that if the program passes *this* series of tests, the hypothesis under test is probably the wrong one.

Enhancing the test case from the story

To achieve this, we might:

- Look for a potentially promising technique
- Work up a starting example of this type of test that appears relevant to the failure under consideration
- Try out the test
  - If you get the failure this simply, you can stop
  - Otherwise, polish the test
    - > Consider the strengths of this class of test
    - > Stretch the test on the attributes not normally emphasized by this technique.

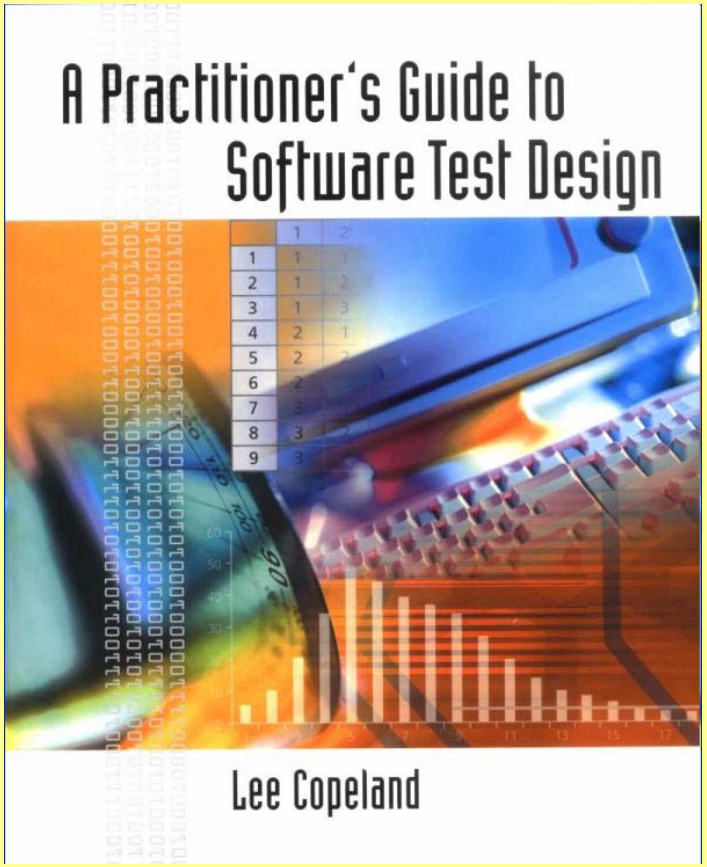
Test Design: Some Readings

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

Kaner, C. (2003) "What is a good test case?" http://www.testingeducation.org/a/testcase.pdf

Whittaker, "What is testing? And why is it so hard?" http://www.computer.org/software/so2000/pdf/s1070.pdf

Whittaker & Atkin, Software Engineering is not Enough, http://www.sisecure.com/pdf/jwsasofteng.pdf



Scenario testing: Developing stories as a vehicle for achieving relevance

Scenario Testing: Some Readings

Berger, Bernie (2001) "The dangers of use cases employed as test cases," STAR West conference, San Jose, CA. www.testassured.com/docs/Dangers.htm. accessed March 30, 2003

Buwalda, Hans (2000a) "The three holy grails of test development," presented at EuroSTAR conference.

Buwalda, Hans (2000b) "Soap Opera Testing," presented at International Software Quality Week Europe conference, Brussels.

Collard, R. (1999, July) "Developing test cases from use cases", Software Testing & Quality Engineering, available at www.stickyminds.com.

Kaner, C. (2003) An introduction to scenario testing, http://www.testingeducation.org/articles/scenario_intro_ver4.pdf

Scenario testing

The ideal scenario has several characteristics:

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

Why use scenario tests?

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

Scenarios

Designing scenario tests is much like doing a requirements analysis, but is not requirements analysis. They rely on similar information but use it differently.

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

Risks of scenario testing

Other approaches are better for testing early, unstable code.

- A scenario is complex, involving many features. If the first feature is broken, the rest of the test can't be run. Once that feature is fixed, the next broken feature blocks the test.
- Test each feature in isolation before testing scenarios, to efficiently expose problems as soon as they appear.

Scenario tests are not designed for coverage of the program.

- It takes exceptional care to cover all features or requirements in a set of scenario tests. Statement coverage simply isn't achieved this way.

Reusing scenarios may lack power and be inefficient

- Documenting and reusing scenarios seems efficient because it takes work to create a good scenario.
- Scenarios often expose design errors but we soon learn what a test teaches about the design.
- Scenarios expose coding errors because they combine many features and much data. To cover more combinations, we need new tests.
- Do regression testing with single-feature tests or unit tests, not scenarios.

Sixteen ways to create good scenarios

- Write life histories for objects in the system. How was the object created, what happens to it, how is it used or modified, what does it interact with, when is it destroyed or discarded?
- List possible users, analyze their interests and objectives.
- Consider disfavored users: how do they want to abuse your system?
- List system events. How does the system handle them?
- List special events. What accommodations does the system make for these?
- List benefits and create end-to-end tasks to check them.
- Look at the specific transactions that people try to complete, such as opening a bank account or sending a message. What are all the steps, data items, outputs, displays, etc.?
- What forms do the users work with? Work with them (read, write, modify, etc.)
- Interview users about famous challenges and failures of the old system.
- Work alongside users to see how they work and what they do.
- Read about what systems like this are supposed to do. Play with competing systems.
- Study complaints about the predecessor to this system or its competitors.
- Create a mock business. Treat it as real and process its data.
- Try converting real-life data from a competing or predecessor application.
- Look at the output that competing applications can create. How would you create these reports / objects / whatever in your application?
- Look for sequences: People (or the system) typically do task X in an order. What are the most common orders (sequences) of subtasks in achieving X?