A Tutorial in Exploratory Testing

April 2008

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 Grants EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers" and CCLI-0717613 "Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing." 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, Rebecca Fiedler, and Mike Kelly.

Conference Abstract

I coined the phrase "exploratory testing" 24 years ago, to describe a style of skilled work that was common in Silicon Valley. Naturally, the concept has evolved since then, primarily as a way of focusing on how testers learn about the product under test and its risks throughout the product's lifecycle.

All testers do exploratory testing. Some do it more deliberately and in intentionally skilled ways. This tutorial considers both the ideas of exploration and several of the skills you can develop and tools you can use to do it better.

Participants will learn:

- Using heuristics to support rapid learning about a product and its risks
- Mining source documents, such as specifications, that are incomplete, out of date, but useful for guiding the investigation of the program
- Splitting effort between tests worth trying once and tests that turn out to be worth keeping, documenting and/or automating.

About these materials

- As always with my tutorials and talks, there are more slides than we will actually get through.
- We'll pick what we actually do based on your interests and questions.
- The slides for the keynote on risk-based testing supplement these slides.
- The lectures at www.testingeducation.org/BBST provide additional (free) explanation of most of these slides.

Outline

- An opening contrast: Scripted testing
- The nature of testing
- The other side of the contrast: Exploration
- Exploratory testing: Learning
- Exploratory testing: Design
- Exploratory testing: Execution
- Exploratory testing: Interpretation
- Exploratory testing after 24 years

Scripted testing

A script specifies

- the test operations
- the expected results
- the comparisons the human or machine should make

These comparison points are

- useful, but fallible and incomplete, criteria for deciding whether the program passed or failed the test

Scripts can control

- manual testing by humans
- automated test execution or comparison by machine

Key benefits of scripts

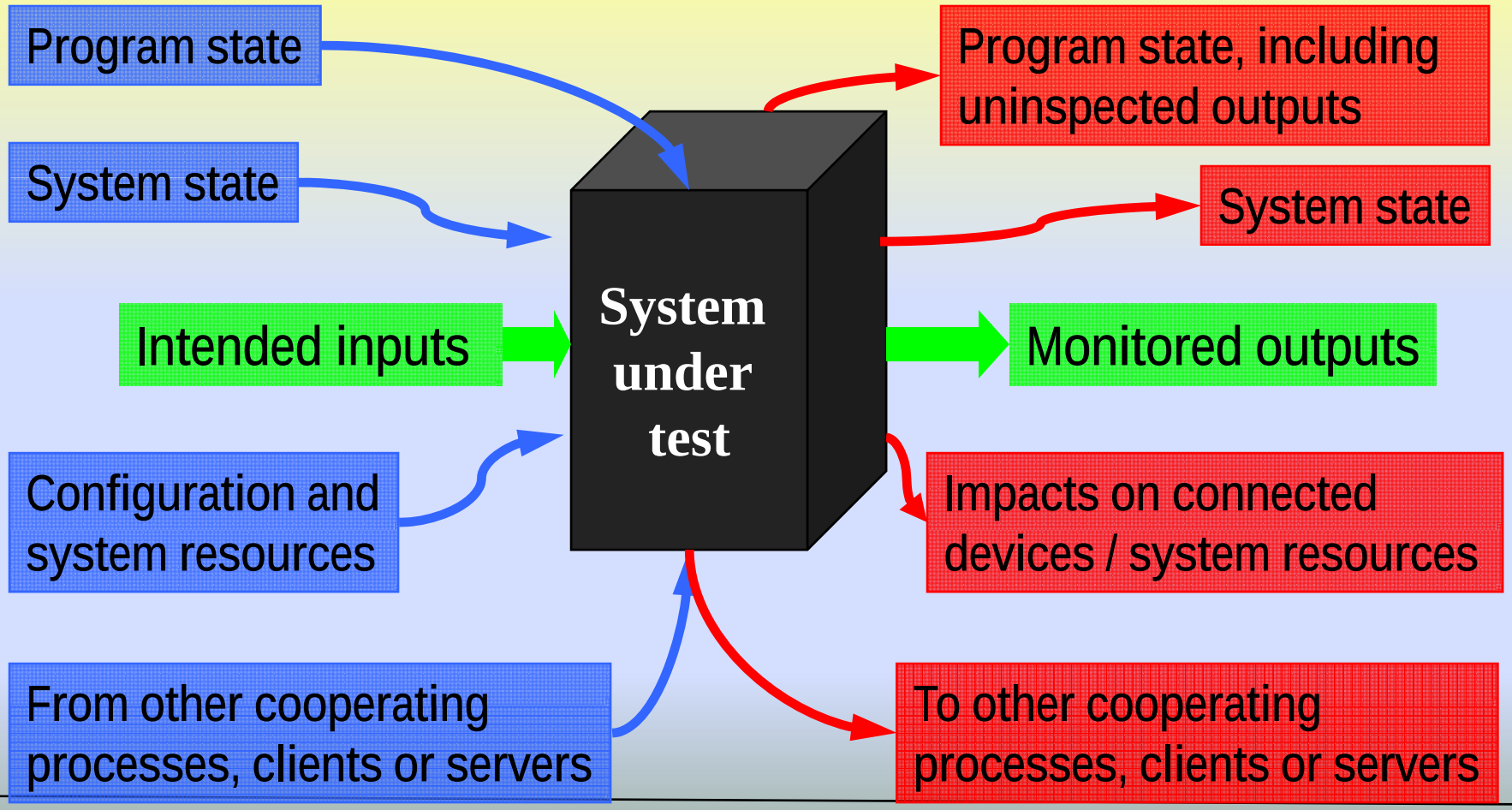
Scripts require a big investment. What do we get back?

The scripting process provides *opportunities* to achieve several key benefits:

- Careful thinking about the design of each test, optimizing it for its most important attributes (power, credibility, whatever)
- Review by other stakeholders
- Reusability
- Known comprehensiveness of the set of tests
- If we consider the set sufficiently comprehensive, we can calculate as a metric the percentage completed of these tests.

Problem with scripts: Programs fail in many ways

Based on notes from Doug Hoffman



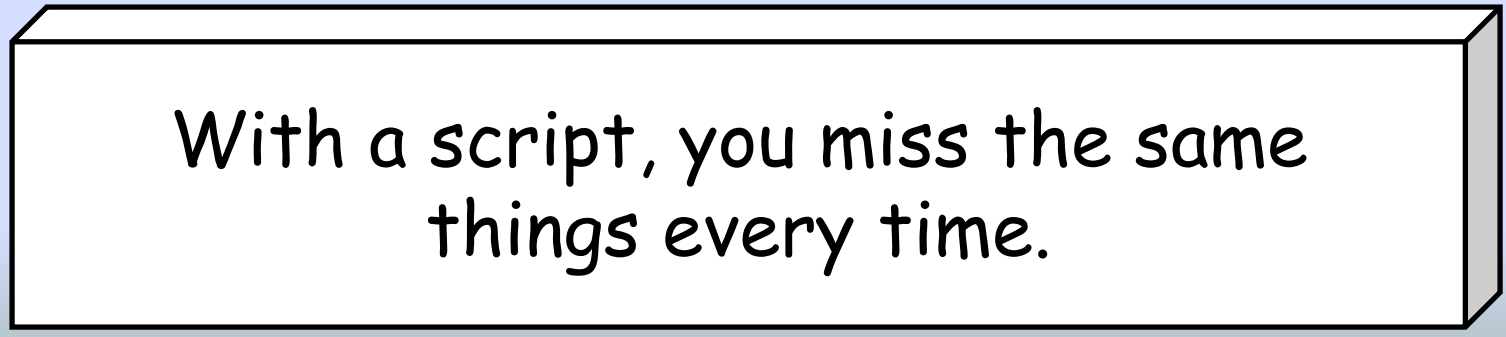
Scripts are hit and miss …

People are finite capacity information processors

- Remember our demonstration of inattentional blindness
- We pay attention to some things
  - and therefore we do NOT pay attention to others
  - Even events that "should be" obvious will be missed if we are attending to other things.

Computers focus only on what they are programmed to look at:

- They are inattentionally blind by design



Time sequence in scripted testing

- Design the test early
- Execute it many times later
- Look for the same things each time

The high-cognitive work in this sequence is done during test design, not during test execution.

Risk profiles evolve over time

Specifying the full set of tests at the start of the project is an invitation to failure:

- The requirements / specifications are almost certain to change as the program evolves
- Different programmers tend to make different errors. (This is a key part of the rationale behind the PSP.) A generic test suite that ignores authorship will overemphasize some potential errors while underemphasizing others.
- The environment in which the software will run (platform, competition, user expectations, new exploits) changes over time.

Time sequence in scripted testing

- Design the test early
- Execute it many times later
- Look for the same things each time

- The earlier you design the tests, the less you understand the program and its risk profile
  - ***And thus, the less well you understand what to look at***

The scripted approach means the test stays the same, even thought the risk profile is changing.

Cognitive sequence in scripted testing

The smart test designer

- who rarely runs the tests

designs the tests for **the cheap tester**

- who does what the designer says to do
- and looks for what the designer says to look for
- time and time again, independently of the risk profile.

This is very cost-effective

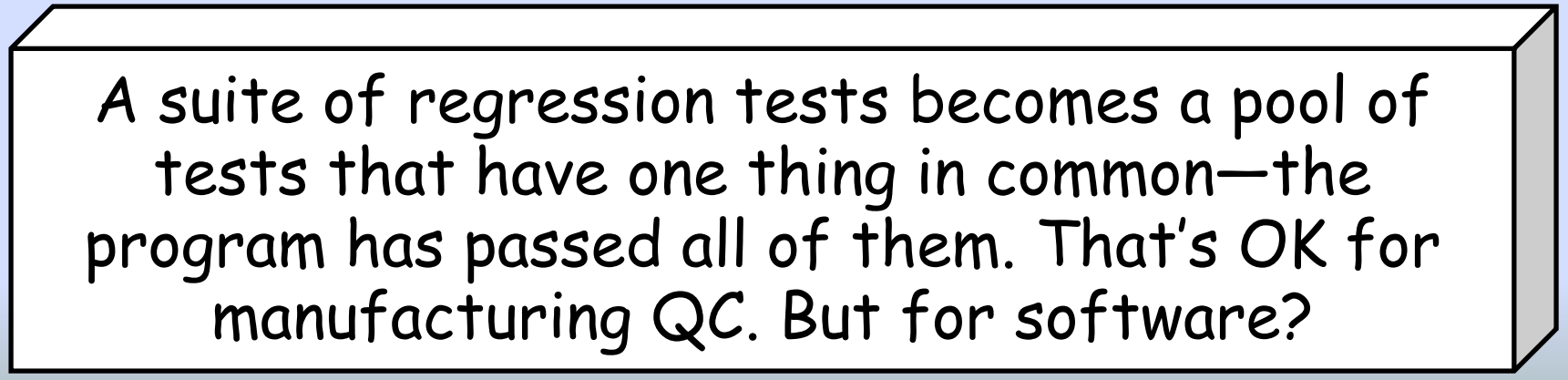
- if the program has no bugs (or only bugs clearly covered in the script)

But what if your program has unexpected bugs?

Who is in a better position to spot changes in risk or to notice new variables to look at?

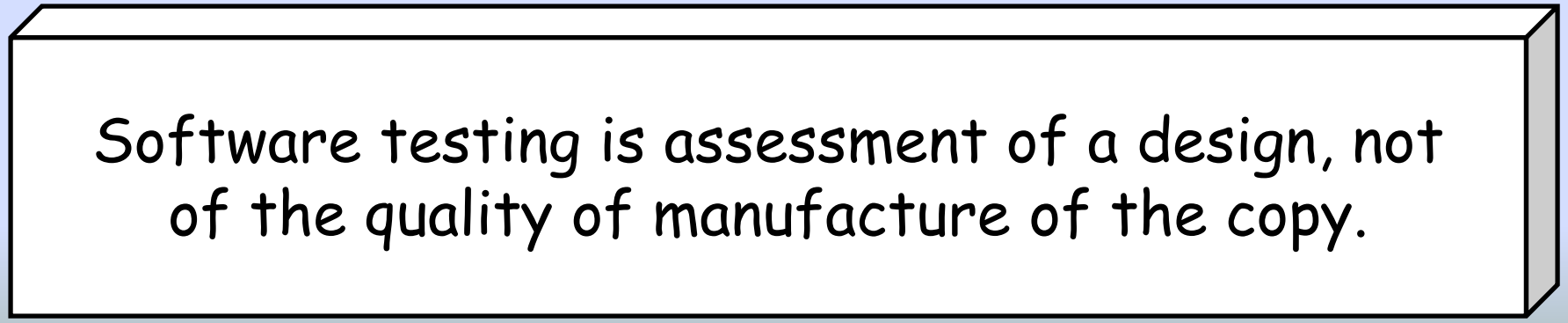
Analogy to Manufacturing QC

- Scripting makes a lot of sense because we have:
  - 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



Analogy to Design QC

- The difference between manufacturing defects and design defects is that:
  - A manufacturing defect appears in an individual instance of the product
  - A design defect appears in every instance 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



Manufacturing versus services

Peter Drucker, Managing in the Next Society, stresses that we should manufacture remotely but provide services locally.

The local service provider is more readily available, more responsive, and more able to understand what is needed.

Most software engineering standards (such as the DoD and IEEE standards) were heavily influenced by contracting firms—outsourcers.

If you choose to outsource development, **of course** you should change your practices to make them look as much like manufacturing as possible.

But is the goal to support outsourcing?

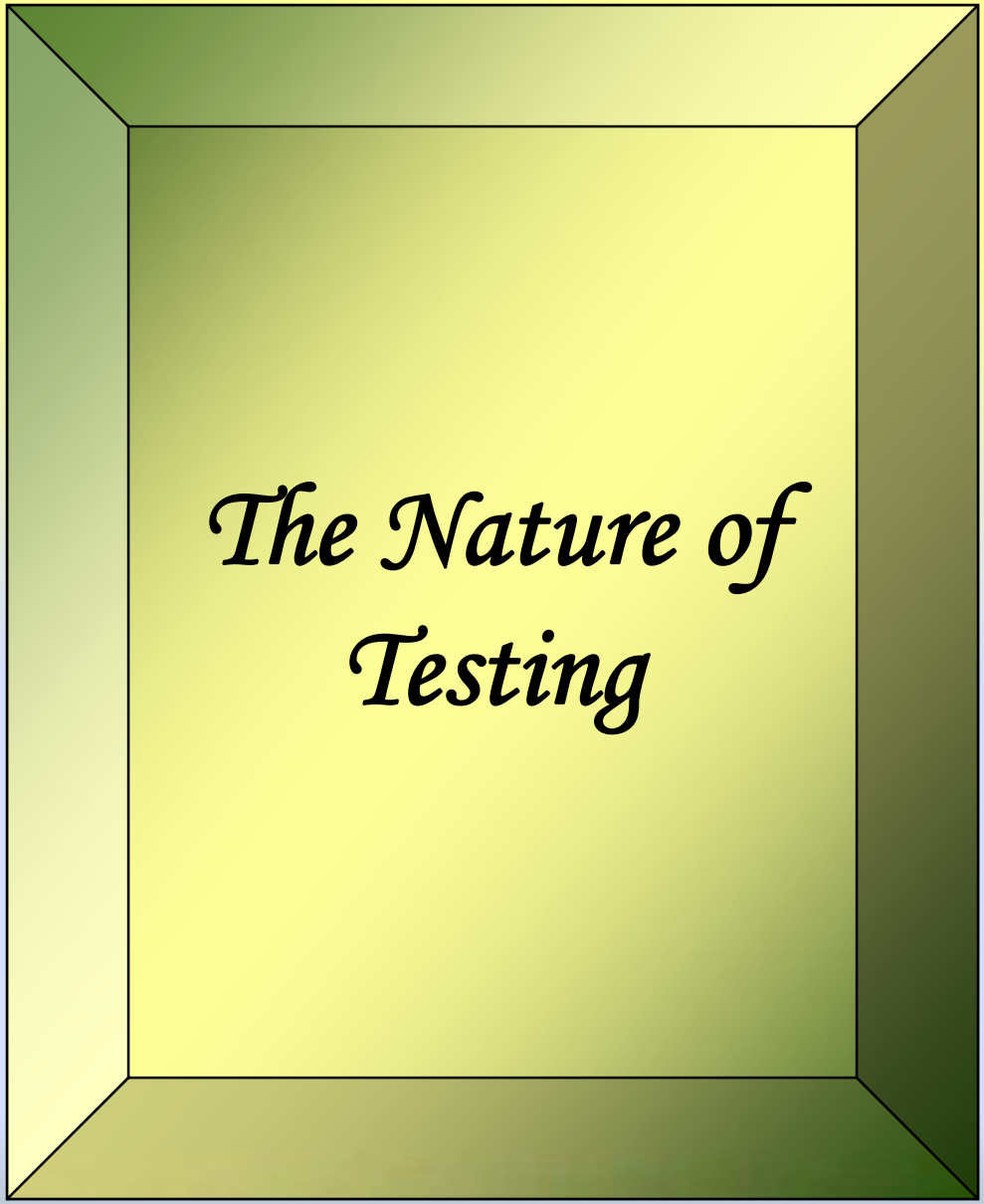
Unless you are the outsource service provider, scripting is probably an industry worst practice for design QC.

What we need for design…

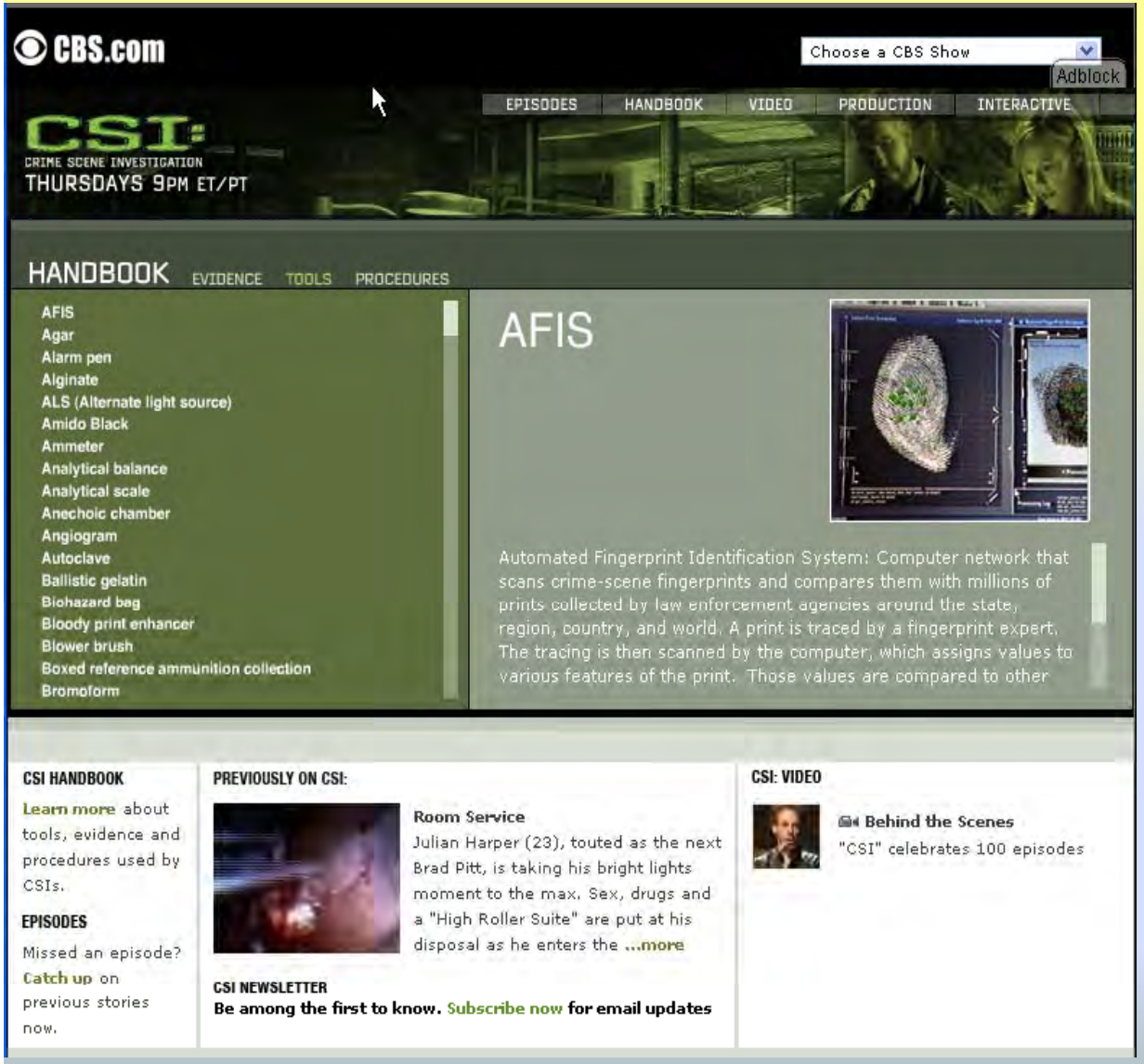
Is a constantly evolving set of tests

- 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



Testing is 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.

Imagine …

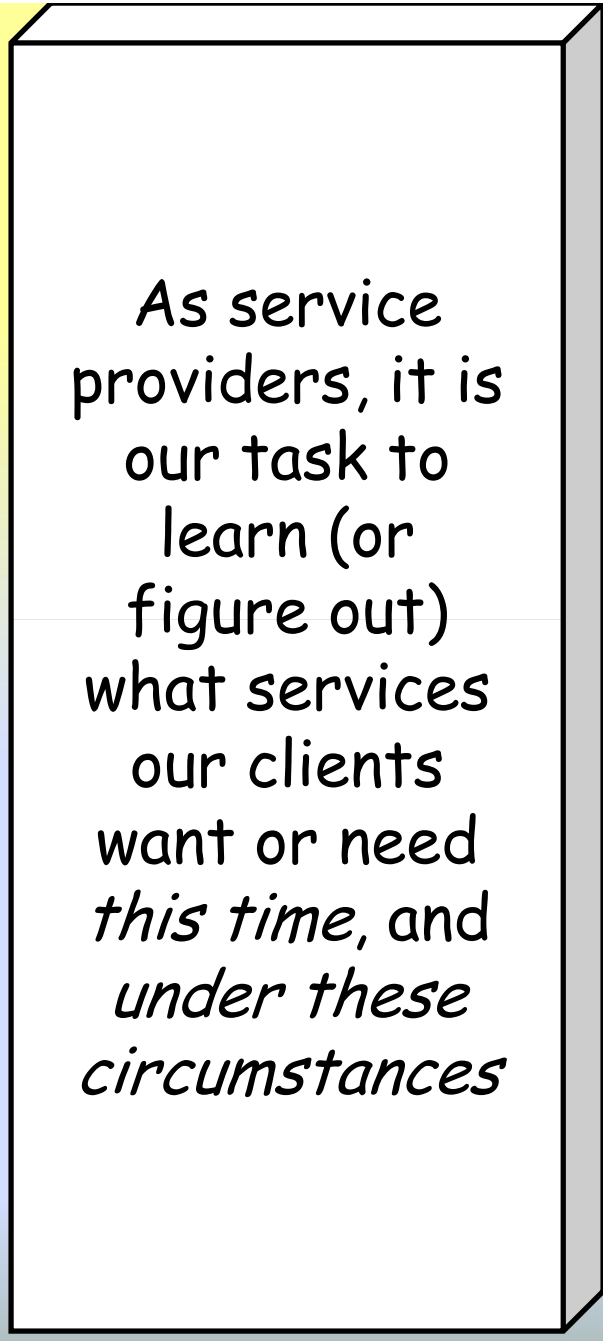
Imagine crime scene investigators

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

How effective do you think they would be?

Testing is always done within a context

- We test in the face of harsh constraints
  - Complete testing is impossible
  - Project schedules and budget are finite
  - Skills of the testing group are limited
- Testing might be done before, during or after a release.
- Improvement of product or process might or might not be an objective of testing.
- We test 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 do they want to mitigate?



Example of context: A thought experiment

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

- Computer game that uses the spreadsheet for occasional tasks like bargaining with another player
- 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
- Late development of a commercial product, to help the project manager decide whether the product is finished
- 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 a failure we're seeking
- How to recognize failure
- How to decide what result variables to attend to
- How to decide what *other* result variables to attend 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.

Testing is always a search for information

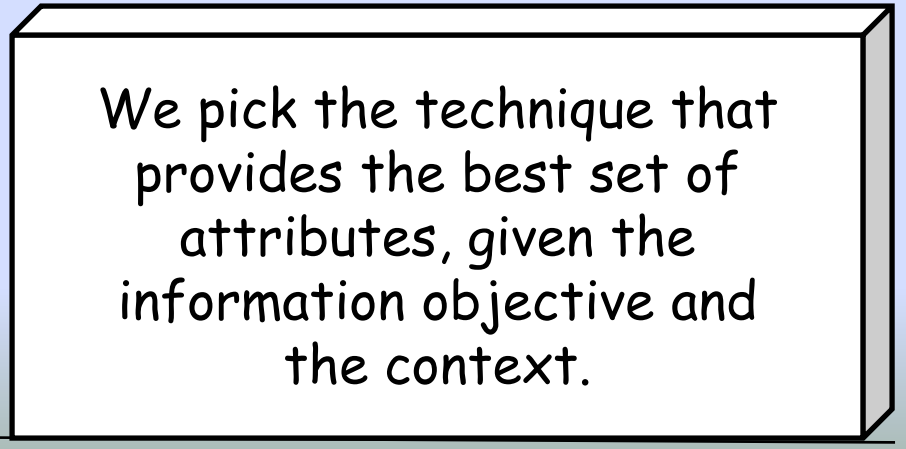
- Find important bugs, to get them fixed
- Assess the quality of the product
- Help managers make release decisions
- Block premature product releases
- Help predict and control product support costs
- Check interoperability with other products
- Find safe scenarios for use of the product
- Assess conformance 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 tools and strategies and will yield different tests, different test documentation and different test results.

Test techniques

A test technique is essentially a recipe, or a model, that guides us in creating specific tests. Examples of common test techniques:

- Function testing
- Specification-based testing
- Domain testing
- Risk-based testing
- Scenario testing
- Regression testing
- Stress testing
- User testing
- All-pairs combination testing
- Data flow testing
- Build verification testing
- State-model based testing
- High volume automated testing
- Printer compatibility testing
- Testing to maximize statement and branch coverage



Examples of test techniques

- **Scenario testing**
  - Tests are complex stories that capture how the program will be used in real-life situations.
- **Specification-based 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.
- **Risk-based 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.

Techniques differ in how to define a good test

Power. When a problem exists, the test will reveal it

Valid. When the test reveals a problem, it is a genuine problem

Value. Reveals things your clients want to know about the product or project

Credible. Client will believe that people will do the things done in this test

Representative of events most likely to be encountered by the user

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

Maintainable. Easy to revise in the face of product changes

Repeatable. Easy and inexpensive to reuse the test.

Performable. Can do the test as designed

Refutability: Designed to challenge basic or critical assumptions (e.g. your theory of the user's goals is all wrong)

Coverage. Part of a collection of tests that together address a class of issues

Easy to evaluate.

Supports troubleshooting. Provides useful information for the debugging programmer

Appropriately complex. As a program gets more stable, use more complex tests

Accountable. You can explain, justify, and prove you ran it

Cost. Includes time and effort, as well as direct costs

Opportunity Cost. Developing and performing this test prevents you from doing other work

Differences in emphasis on different test attributes

- **Scenario testing:**
- complex stories that capture how the program will be used in real-life situations
  - Good scenarios focus on validity, complexity, credibility, motivational effect
  - The scenario designer might care less about power, maintainability, coverage, reusability
- **Risk-based testing:**
- Imagine how the program could fail, and try to get it to fail that way
  - Good risk-based tests are powerful, valid, non-redundant, and aim at high-stakes issues (refutability)
  - The risk-based tester might not care as much about credibility, representativeness, performability—we can work on these after (if) a test exposes a bug

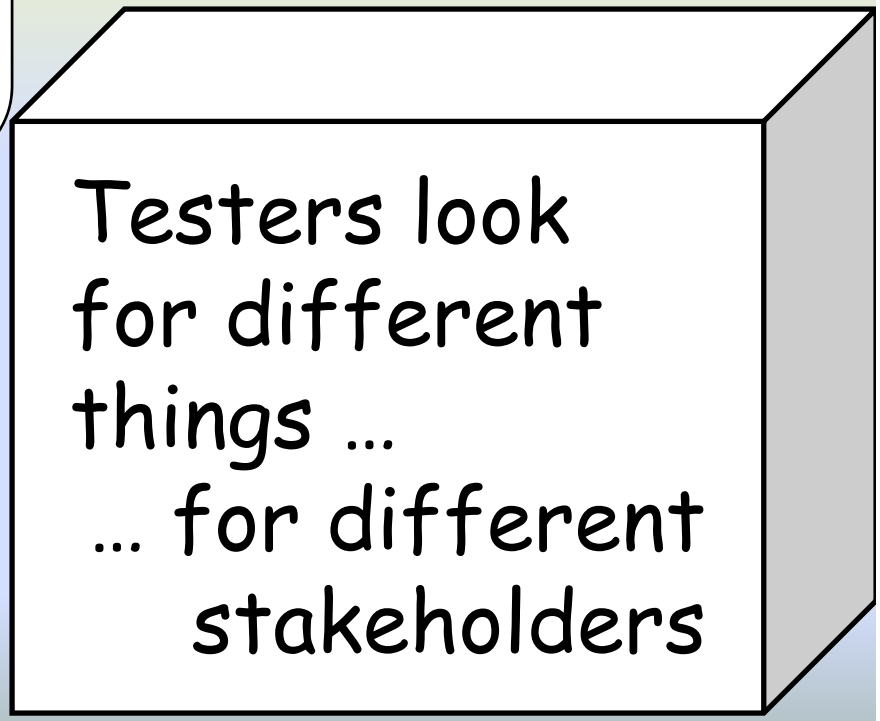
Quality and errors

Quality is value to some person

-- Jerry Weinberg

Under this view:

- Quality is inherently subjective
  - Different stakeholders will perceive the same product as having different levels of quality



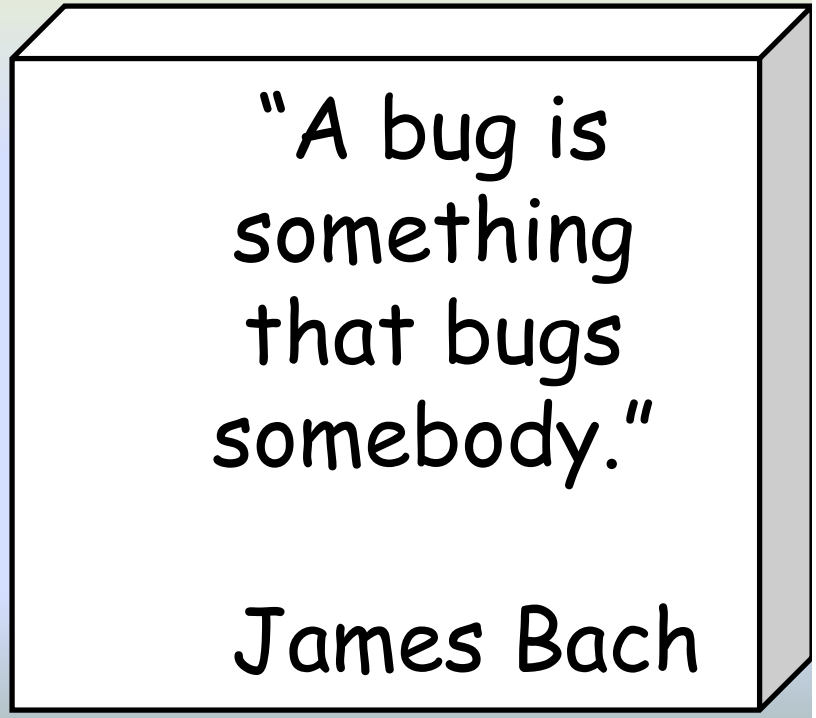
Software error

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.

An error:

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



Reject the "Not My Job" definition of testing

- Testing is not only about doing tasks some programmer can imagine for you or meeting objectives some programmer wishes on you
  - unless that programmer is your primary stakeholder
- The tester who looks only for coding errors misses all the other ways in which a program is of lower quality than it should be.
- Anything that threatens a product's value to a stakeholder with influence threatens quality in a way important to the project.
  - You might be asked to investigate any type of threat, including security, performance, usability, suitability, etc.

Tasks beyond your personal skill set may 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

We design and run tests in order to gain useful information about the product's quality

Test and test case

Think of **a test** as a question that you ask the program.

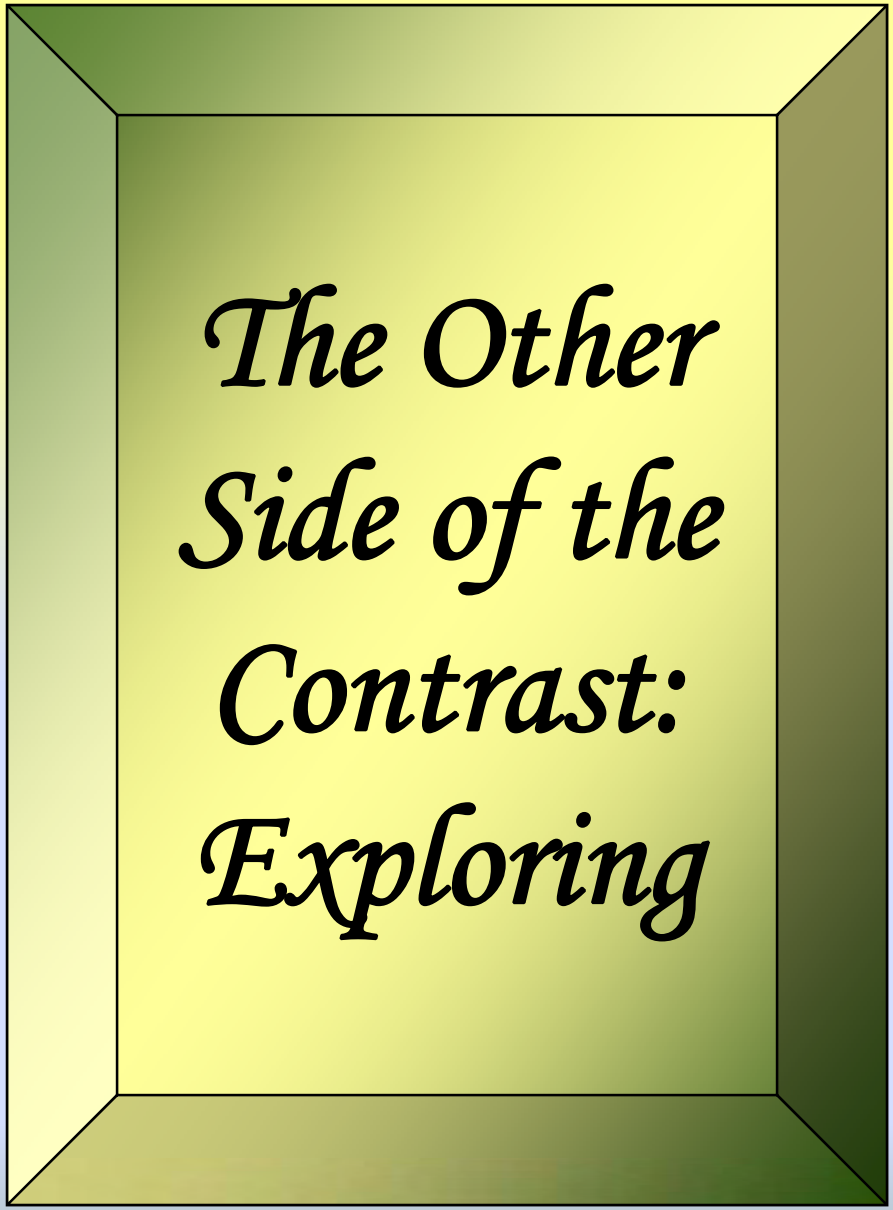
- You run the test (the experiment) in order to answer the question.

A **test case** is a test

- Usually, when we just say "a test", we mean something we do,
- Usually, when we say "test case," we mean something that we have described / documented.

A **test idea** is the thought that guides our creation of a test. For example, "what's the boundary of this variable? Can we test it?" is a test idea.

For our purposes today, the distinction between test and test case is irrelevant, and I will switch freely between the two terms.



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.

Time sequence in exploration

- In contrast with scripting, we:
- Design the test as needed
- Execute the test at time of design or reuse it later
- Vary the test as appropriate, whenever appropriate.

Not scripting doesn't mean not preparing:

- We often design support materials in advance and use them many times throughout testing, such as
  - data sets
  - failure mode lists
  - combination charts.

Unscripted doesn't mean unprepared.
It's about enabling choice, not constraining it.

Cognitive sequence in exploration

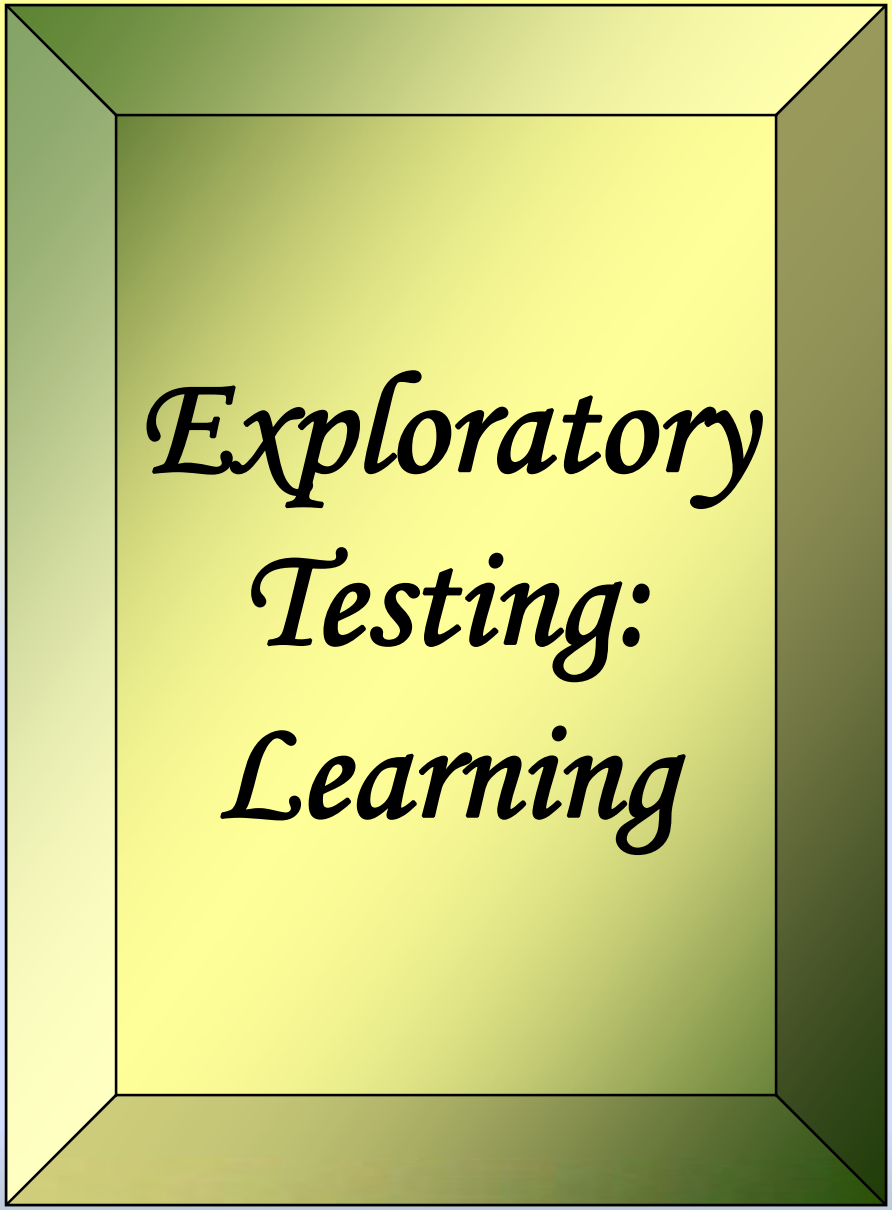
This is the fundamental difference between exploratory and scripted testing.

- **The exploratory tester is always responsible for managing the value of her own time.**
  - At any point in time, this might include:
    - Reusing old tests
    - Creating and running new tests
    - Creating test-support artifacts, such as failure mode lists
    - Conducting background research that can then guide test design

The explorer can do any combination of learning, designing, executing and interpreting at any time.

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**: Doing the test and collecting the results. 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

- **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, data relationship charts, file structures, user tasks, product benefits, FMEA)
- **Question**: Identify missing info, imagine potential sources and potentially revealing questions (interview users, developers, and other stakeholders, use reference materials to supplement answers)
- **Review written sources**: specifications, other authoritative documents, culturally authoritative sources, persuasive sources
- **Try out potentially useful tools**

Examples of learning activities

- **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 learning activities

Create and apply models:

- A model is a simplified representation of a relationship, process or system. The simplification makes some aspects of the thing modeled clearer, more visible, and easier to work with.
- A model is often an expression of something we don't understand in terms of something we (think we) do understand
- All tests are based on models:
  - Many 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)

What are we modeling?

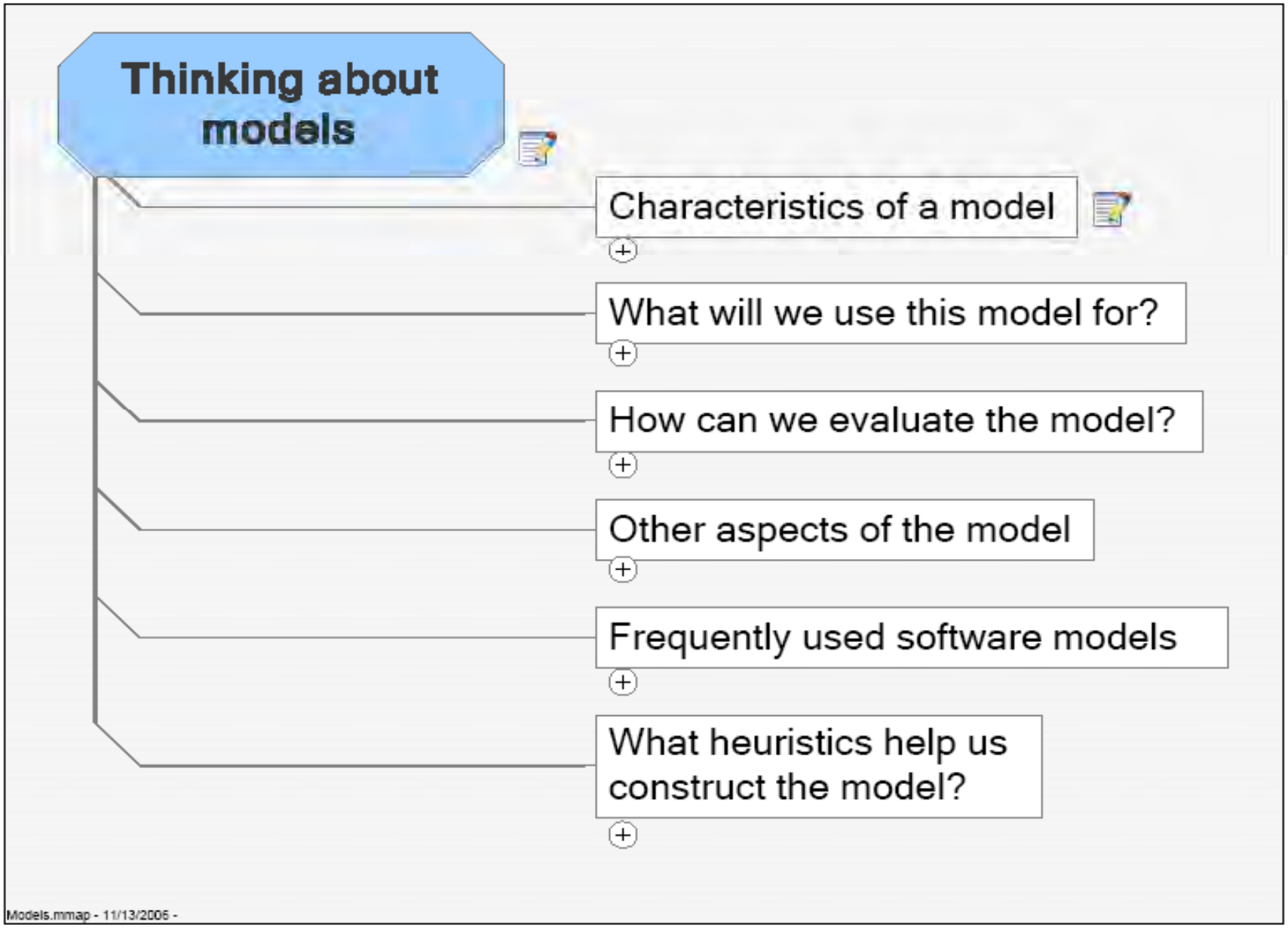
- A physical process emulated, controlled or analyzed by software under test
- A business process emulated, controlled or analyzed by software under test
- Software being emulated, controlled, communicated with or analyzed by the software under test
- Device(s) this program will interact with
- The stakeholder community
- The uses / usage patterns of the product
- The transactions that this product participates in
- The development project
- The user interface of the product
- The objects created by this product

What aspects of them are we modeling?

- Capabilities
- Preferences
  - Competitive analysis
  - Support records
- Focused chronology
  - Achievement of a task or life history of an object or action
- Sequences of actions
  - Such as state diagrams or other sequence diagrams
  - Flow of control
- Flow of information
  - Such as data flow diagrams or protocol diagrams or maps
- Interactions / dependencies
  - Such as combination charts or decision trees
  - Charts of data dependencies
  - Charts of connections of parts of a system
- Collections
  - Such as taxonomies or parallel lists
- Motives
  - Interest analysis
  - Who is affected how, by what?

What makes these models, *models?*

- The representation is simpler than what is modeled: It emphasizes some aspects of what is modeled while hiding other aspects
- You can work with the representation to make descriptions or predictions about the underlying subject of the model
- Using the model is easier or more convenient to work with, or more likely to lead to new insights than working with the original.


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 model of learning

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

This is an adaptation of Anderson/Krathwohl's learning taxonomy. For a summary and links, see http://www.satisfice.com/kaner/?p=14

Focusing on models

- All tests ***are*** based on models
  - But any cognitive or perceptual psychologist will tell you that all perceptions and all judgments are based on models
    - Most of which are implicit

A model of learning

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

This is an adaptation of Anderson/Krathwohl's learning taxonomy. For a summary and links, see http://www.satisfice.com/kaner/?p=14

Focusing on models

- All tests ***are*** based on models
  - But any cognitive or perceptual psychologist will tell you that all perceptions and all judgments are based on models
    - Most of which are implicit
- So the question is,
  - Is it **useful** to focus on discovering, evaluating, extending, and creating models
  - Or are we sometimes better off leaving the models in the background while we focus on the things we are modeling?

Do we make ET impractical if we insist on teaching / working at a high level of cognition or expertise?

Learning from a 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?

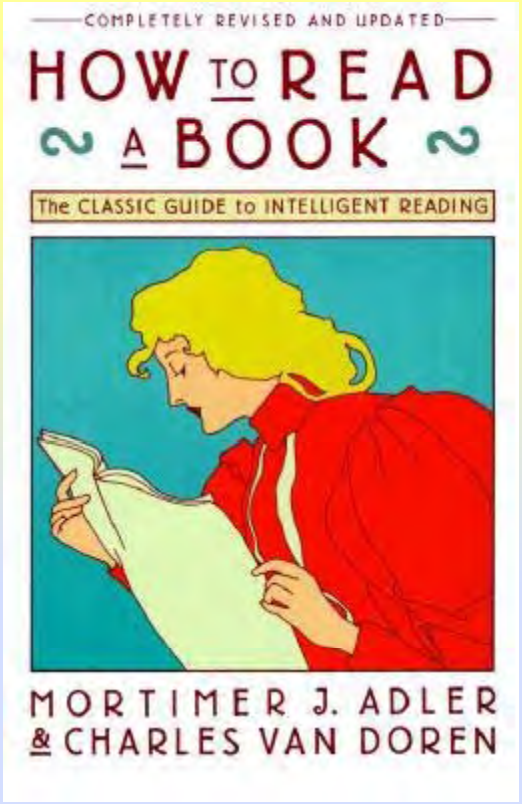
Much of what is written about specification analysis has to do with the specification-in-the-small—interpreting the fine details in one or two pages of text

- 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

Active 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.

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

- **Describe it:** describe its physical attributes (size, shape, etc.) and its functional attributes;
- **Compare it**: What's it similar to? Why do you think so?
- **Associate it:** What other ideas, products, etc. does it bring to mind?
- **Analyze it:** Break it down into its components. How are they related? How do they work together?
- **Apply it:** What can you (or the user) do with it?
- **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.

Active Reading: Cubing (2)

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.

- http://www.itrc.ucf.edu/forpd/strategies/stratCubing.html
- http://www.uhv.edu/ac/research/prewrite/generateideas.pdf
- http://www.humboldt.edu/~tdd2/Cubing.htm

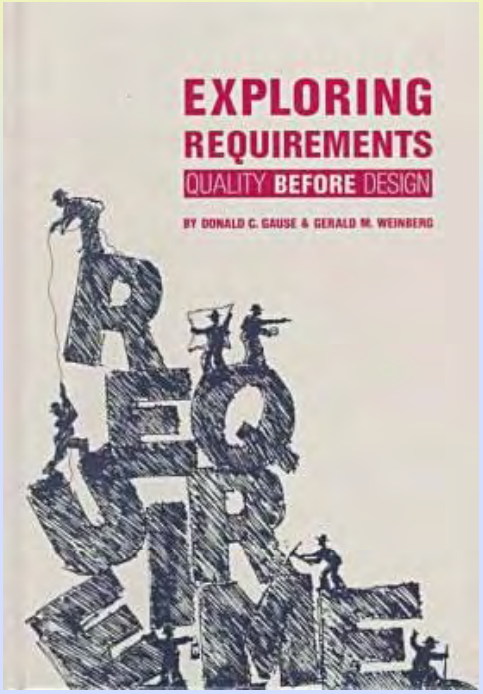
Asking questions

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)

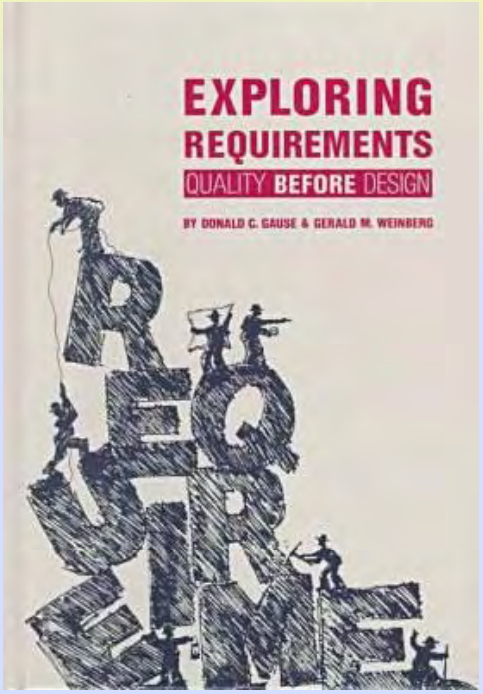


Gause / Weinberg is a superb source for context-free questions

Asking questions (contrasts 2)

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?)

More questions

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?

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?)

More questions

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.)?)

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

The classic context-free questions

Traditional news reporters' questions:

Who? **What?**

When? **Where?**

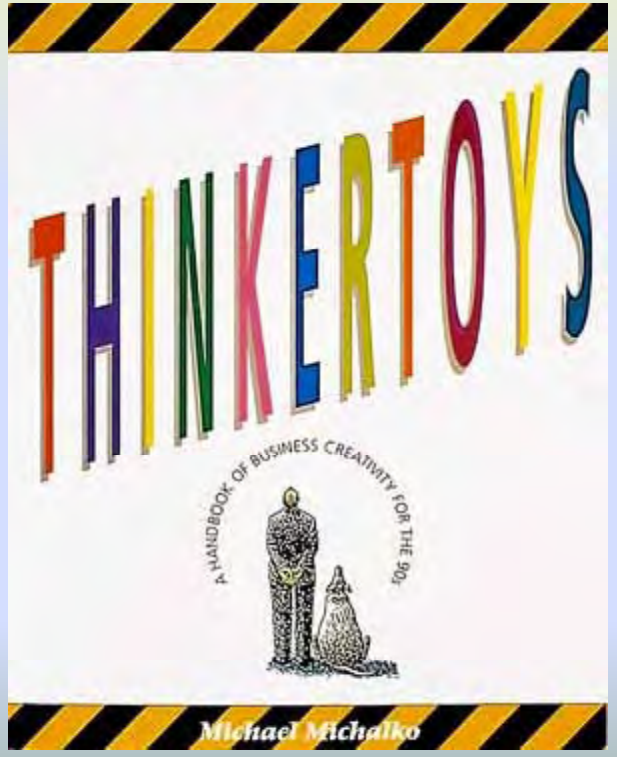
How? **Why?**

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?

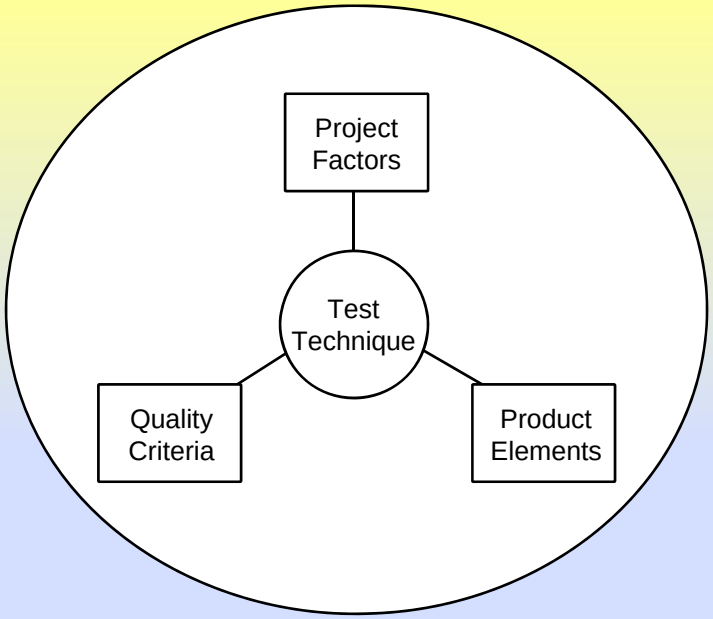
Metaquestions (when interviewing someone for info)

- 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

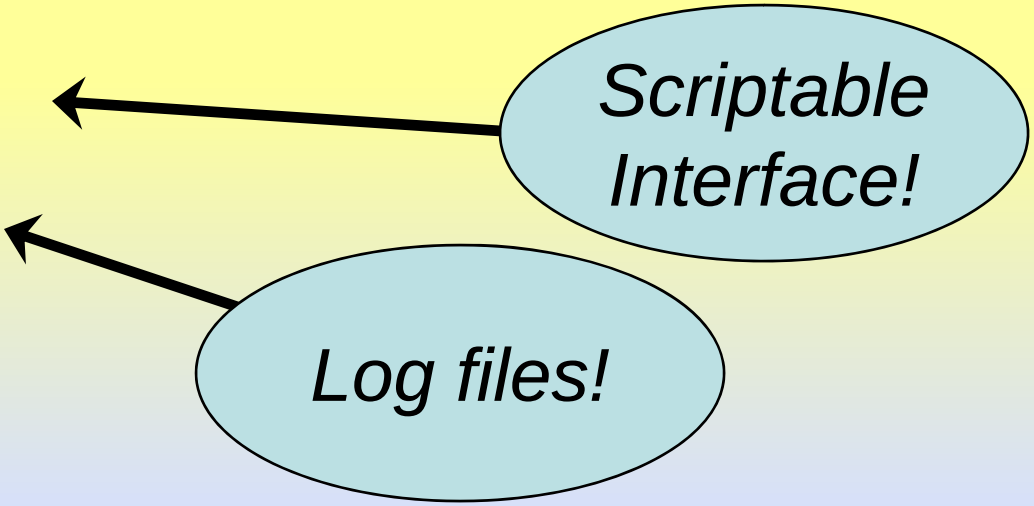
Simplicity

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

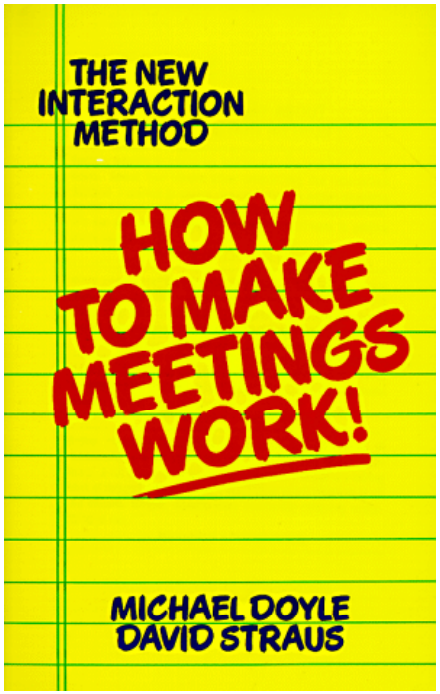
Critiquing specs: Process notes

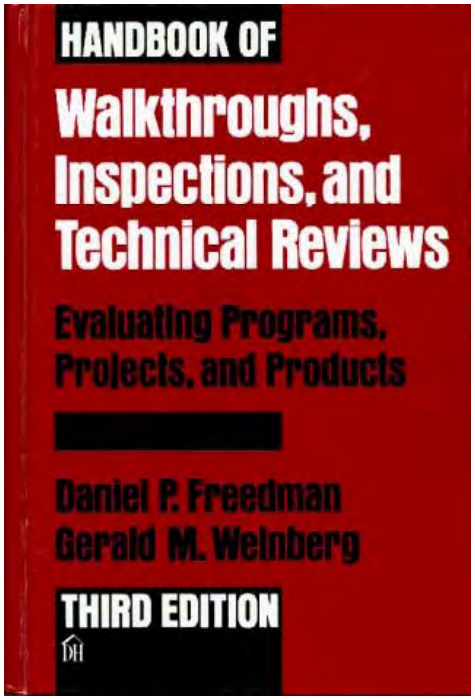
Review meetings

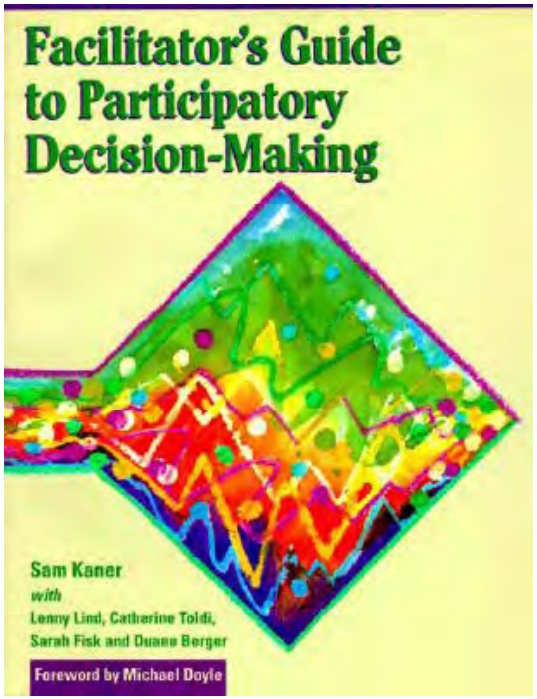
- Test groups often train to facilitate technical reviews

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.

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

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

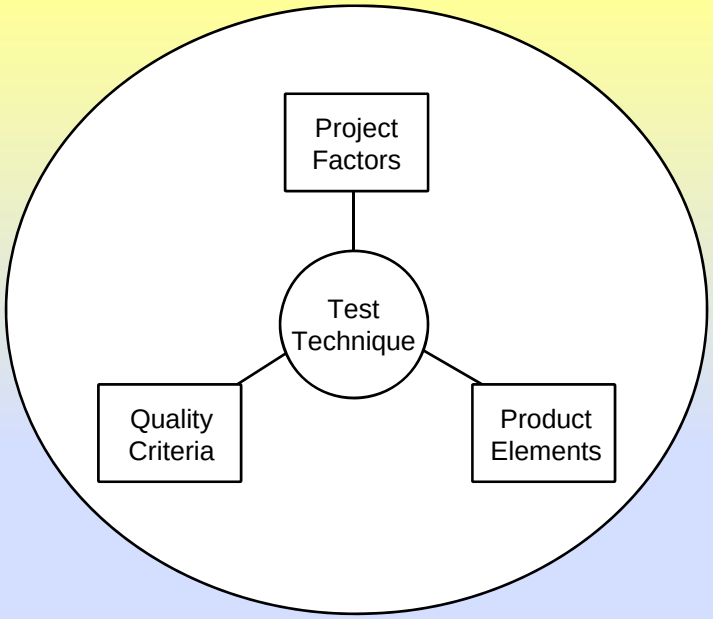
- 10 years of critical peer review by colleagues.
- Several of us have found this a very useful tool for
  - Guiding exploration (see Bach's and Bolton's courses)
  - Structuring a failure mode and effects analysis
    - See Giri Vijayaraghavan & Cem Kaner Bug taxonomies: Use them to generate better 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.

Exploratory testing: Design

- **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:** Doing the test and collecting the results. 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?

Examples of design activities

- 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

Designing test scenarios

1. 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?
2. List possible users, analyze their interests and objectives.
3. Consider disfavored users: how do they want to abuse your system?
4. List system events. How does the system handle them?
5. List special events. What accommodations does the system make for these?
6. List benefits and create end-to-end tasks to check them.
7. 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.?
8. What forms do the users work with? Work with them (read, write, modify, etc.)

Designing test scenarios

9. Interview users about famous challenges and failures of the old system.
10. Work alongside users to see how they work and what they do.
11. Read about what systems like this are supposed to do. Play with competing systems.
12. Study complaints about the predecessor to this system or its competitors.
13. Create a mock business. Treat it as real and process its data.
14. Try converting real-life data from a competing or predecessor application.
15. Look at the output that competing applications can create. How would you create these reports / objects / whatever in your application?
16. 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?

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.

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

Design: Challenge of relevance

- 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?

Design: Challenge of relevance

- 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 **failure mode 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?

How do we map from a failure mode to a test?

Design: Mapping from the failure mode to the test

- Imagine that someone called your company's help desk and complained that the program had failed.
  - They were working in this part of the program
  - And the program displayed some junk on the screen and then crashed
  - They don't know how to recreate the bug but that's no surprise because they have no testing experience.

How would you troubleshoot this report?

In terms of the skills you use, working from a failure mode to a test is like trying to replicate someone else's inadequate report of that failure.

Design: Mapping from the test idea to the test

- Let's create a slightly more concrete version of this 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.
  - The fridge crashed with an unintelligible error message.
- So, how to troubleshoot this problem?
- **First question: What about this error message?**
  - System-level (probably part of the crash, the programmers won't have useful info for us)
  - Application-level (what messages are possible at this point?)
  - **This leads us to our first series of tests:** Try to recreate every error message that can come from requesting a shopping list. Does this testing suggest anything?

Design: Evolving the test case from the story

- **Second question:** What makes a system crash?
  - Data overflow (too much stuff in the fridge?)
  - Wild pointer ("grunge" accumulates because we've used the fridge too long without rebooting?)
  - Stack overflow (what could cause a stack overflow? Ask the programmers.)
  - Unusual timing condition? (Can we create a script that lets us adjust timing of our input to the fridge?)
  - Unusual collection of things in the fridge?
- If you had a real customer who reported this problem, you MIGHT be able to get some of this information from them. But in risk-based testing, you don't have that customer. You just have to work backwards from a hypothetical failure to the conditions that might have produced it. Each set of conditions defines a new test.

How to 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. (For example, if you think it might be a timing problem, what techniques help you focus on timing issues?)
  - Turn the failure into a story and gradually evolve the story into something you can test from. (This is what we did with Joe and the Fridge. A story is easier for some people to work with than a technologically equivalent, but inhuman, description of a failure.
- There are no guarantees in this, but you get better at it as you practice, and as you build a broader inventory of techniques.

More on design

The more test techniques you know, the better your set of choices for mapping test ideas to tests.

This week's keynote on risk-based testing describes a variety of quicktests and other test techniques that are useful for exploratory test design.

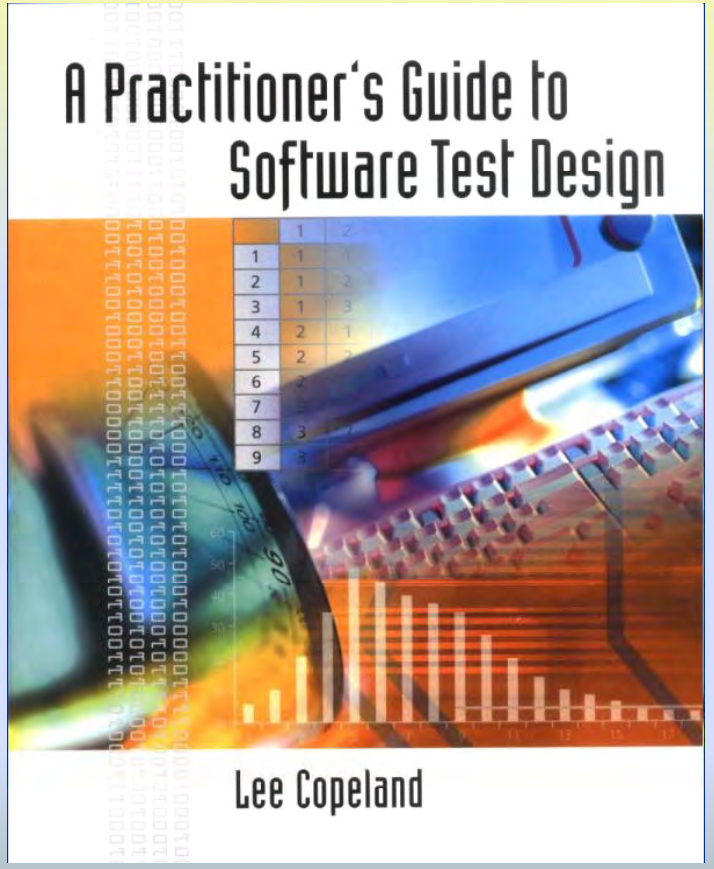
Test Design: Some Readings

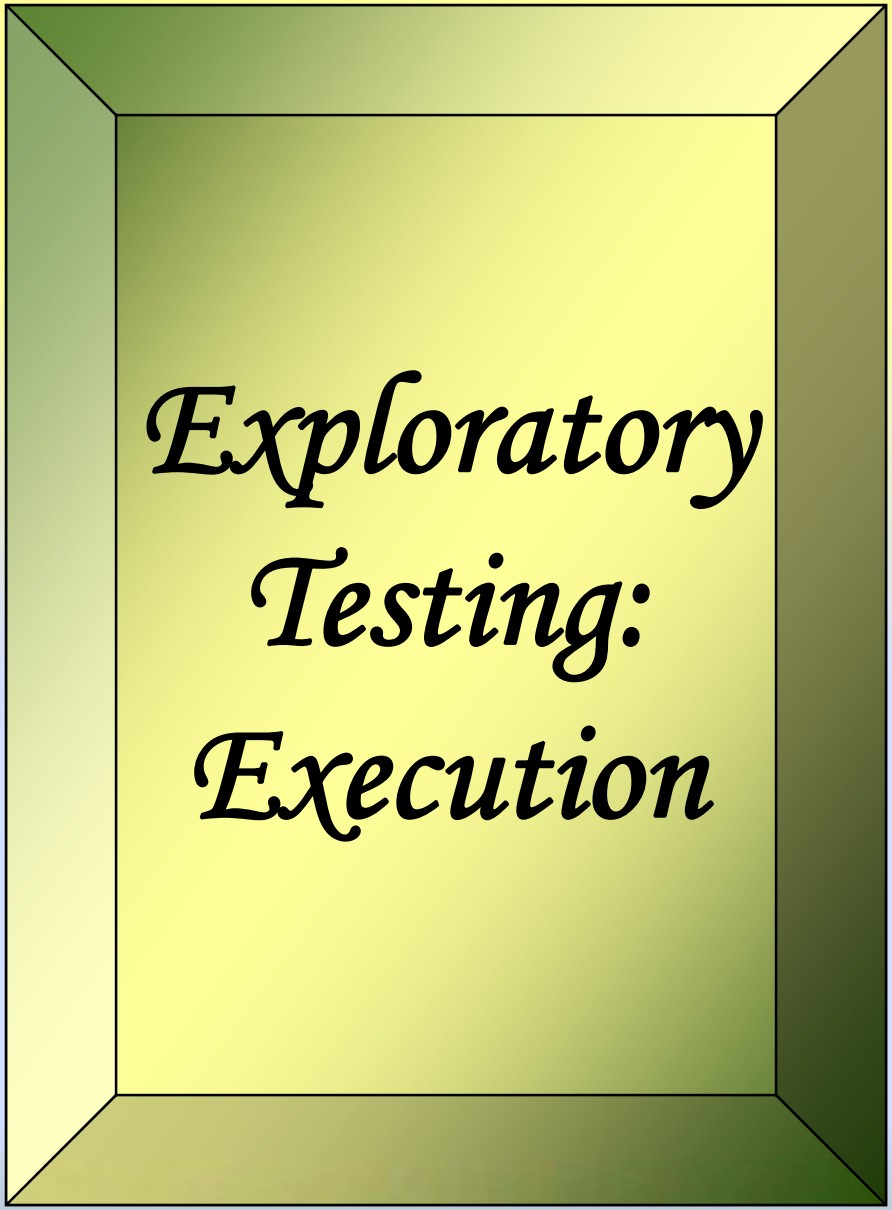
Kaner, Bach & Pettichord, "Testing Techniques" in *Lessons Learned in Software Testing*.

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





Exploratory testing: Execution

- **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**: Doing the test and collecting the results. 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?

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

Scripted execution

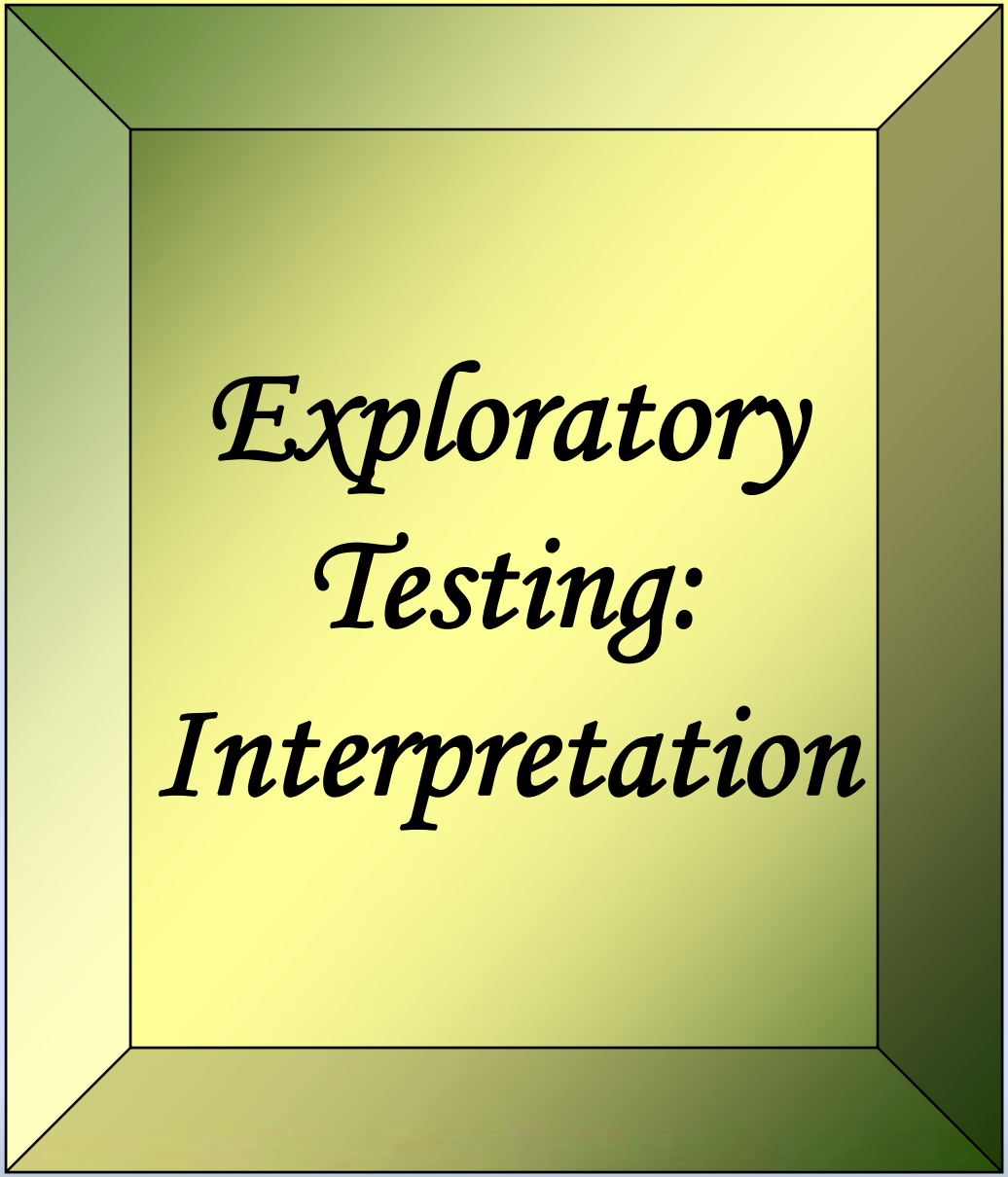
	COGNITIVE PROCESSES					
KNOWLEDGE DIMENSIONS	Remember	Understand	Apply	Analyze	Evaluate	Create
Facts						
Concepts						
Procedures						
Cognitive strategies						
Models						
Skills						
Attitudes						
Metacognition						

The individual contributor (tester rather than “test planner” or manager)

Exploratory execution

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

The individual contributor (tester rather than “test planner” or manager)



Exploratory testing: Interpreting

- 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: Doing the test and collecting the results. 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?

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: Oracle heuristics

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.

Interpretation: Oracle heuristics

Consistent with our Image: Behavior consistent with an image that the organization wants to project.

Consistent with Claims: Behavior consistent with documentation or ads.

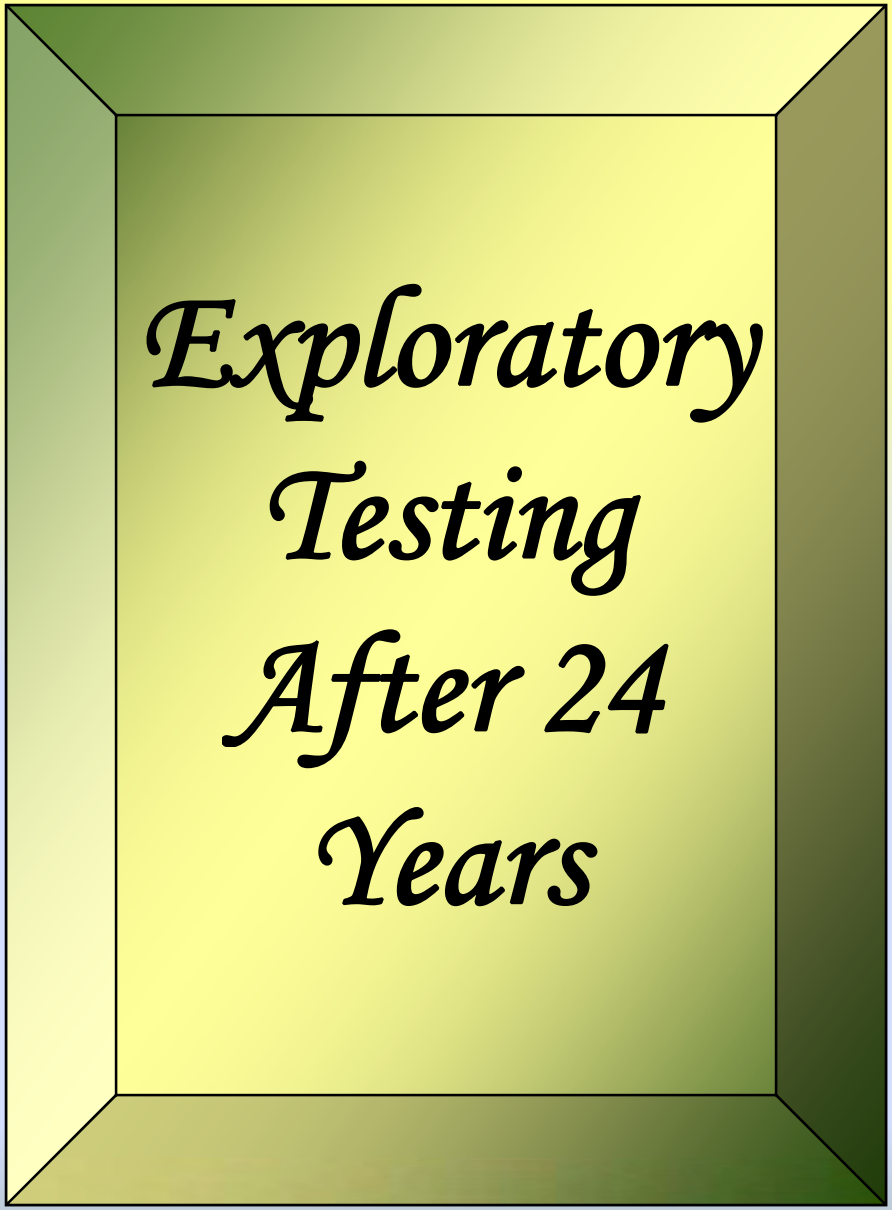
Consistent with Specifications or Regulations: Behavior consistent with claims that must be met.

Consistent with User's Expectations: Behavior consistent with what we think users want.

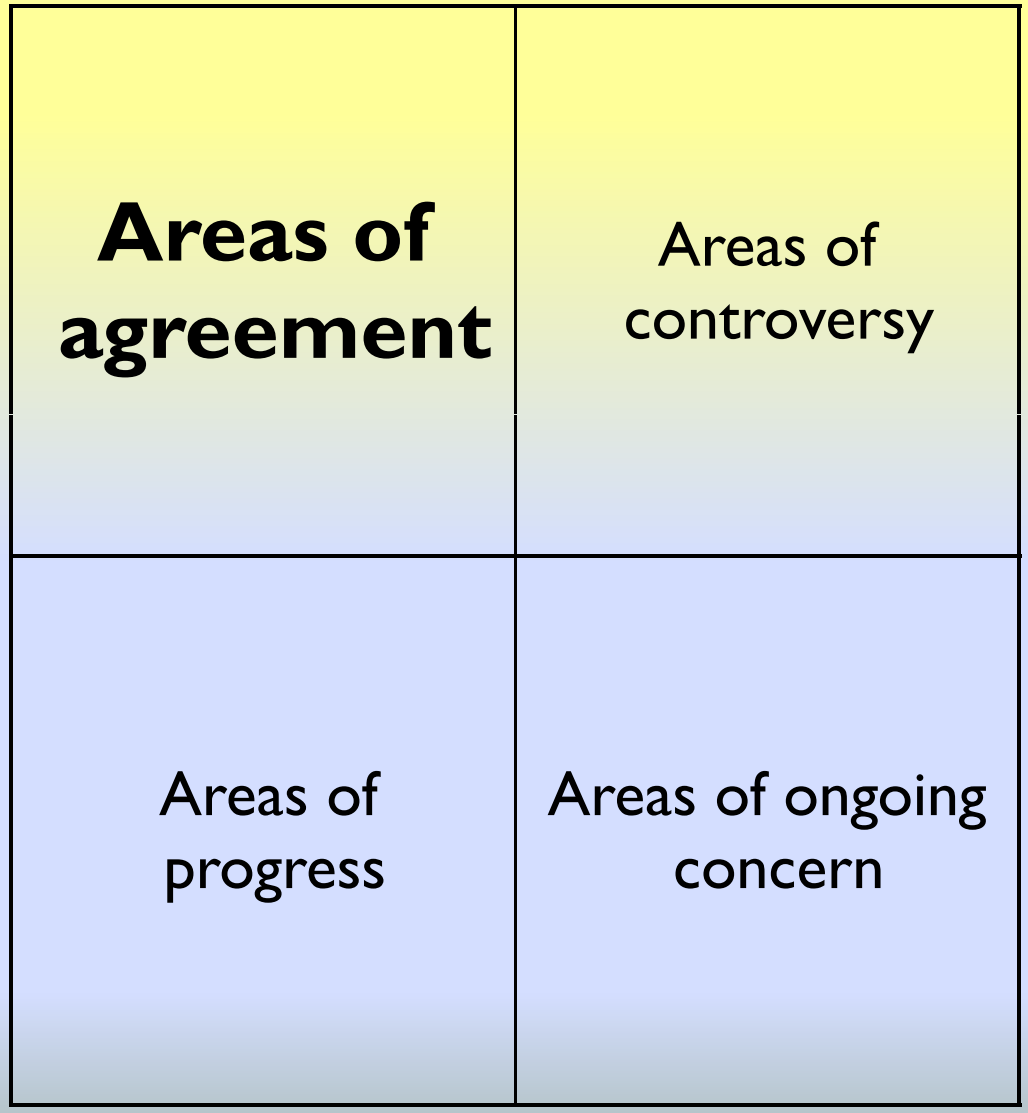
Consistent with Purpose: Behavior consistent with apparent purpose.

Another set of activity descriptions

- Jon Bach, Mike Kelly, and James Bach are working on a broad listing / tutorial of ET activities. See Exploratory Testing Dynamics at http://www.quardev.com/whitepapers.html
- We reviewed preliminary drafts at the Exploratory Testing Research Summit (spring 2006) and Consultants Camp 2006 (August), looking specifically at teaching issues.
- This short paper handout provides an outline for what should be a 3-4 day course. It's a stunningly rich set of skills.
- In this abbreviated form, the lists are particularly useful for audit and mentoring purposes, to highlight gaps in your test activities or those of someone whose work you are evaluating.



Exploratory testing after 24 years

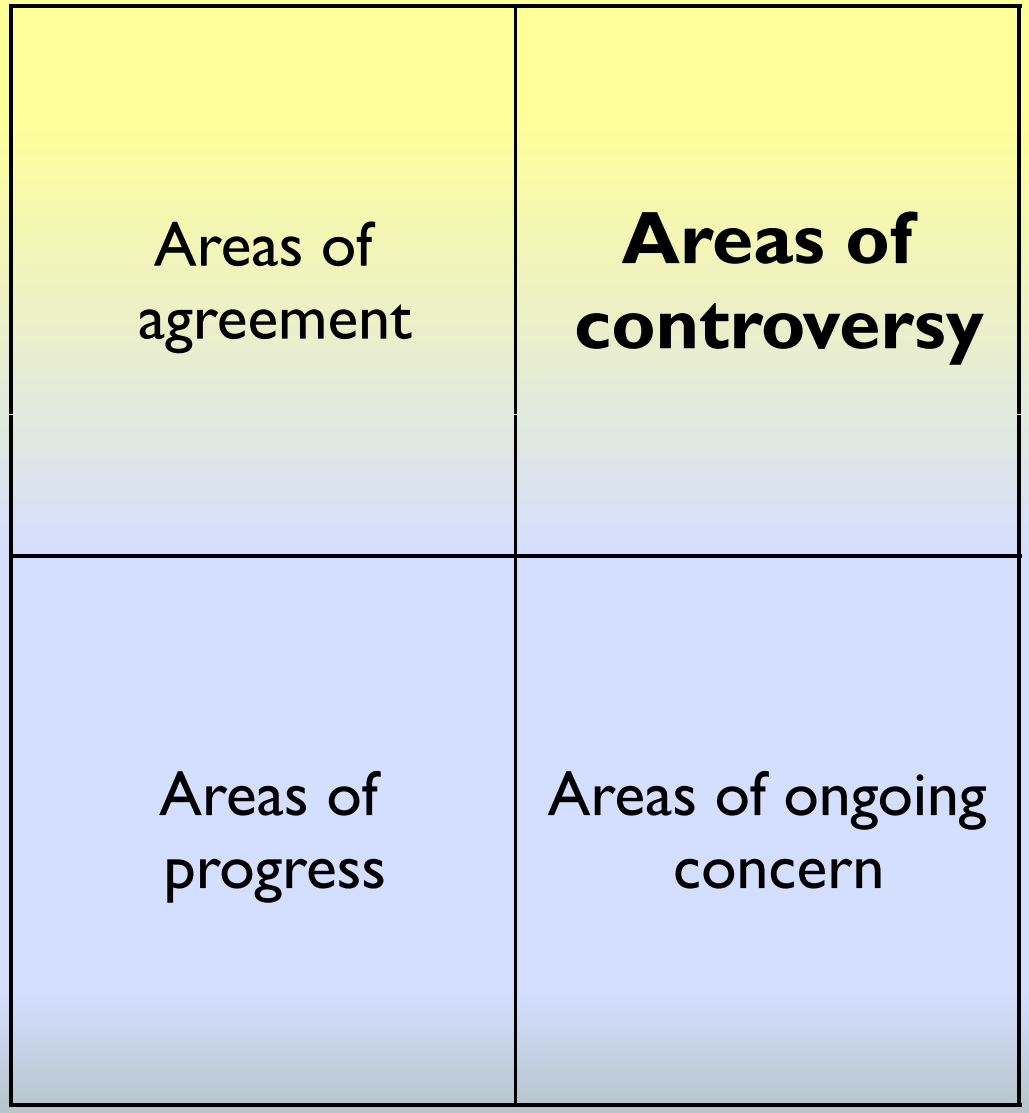


Areas of agreement*

- Definitions
- Everyone does ET to some degree
- ET is an approach, not a technique
- ET is the response (the antithesis) to scripting
  - But a piece of work can be a blend, to some degree exploratory and to some degree scripted

- **Agreement among the people who agree with me (many of whom are sources of my ideas). This is a subset of the population of ET-thinkers who I respect, and a smaller subset of the pool of testers who feel qualified to write about ET. (YMMV)**

Exploratory testing after 24 years



Areas of controversy

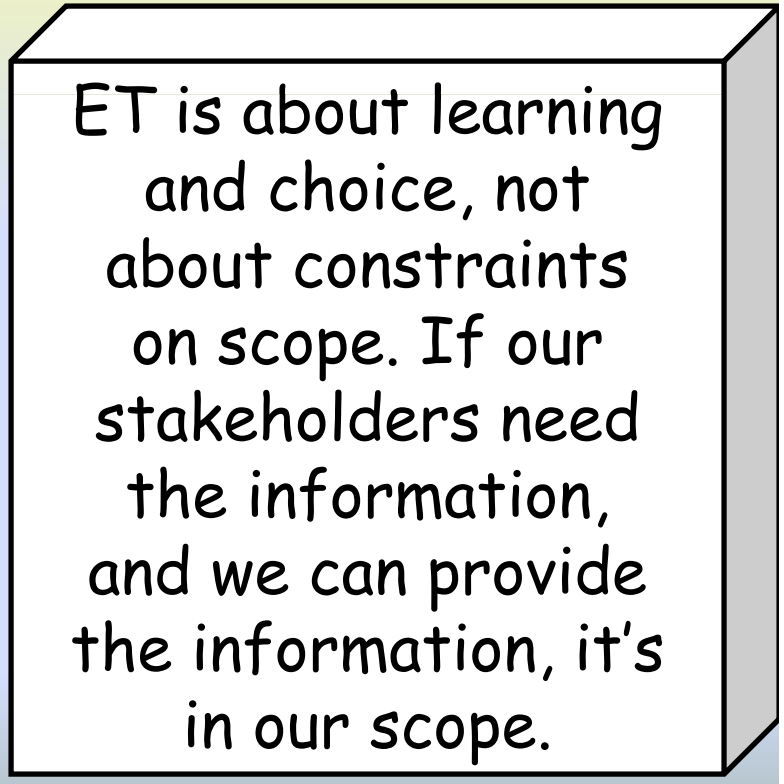
ET is not quicktesting

- A quicktest (or an "attack") is a cheap test that requires little preparation, knowledge or time to perform.
- A quicktest is a technique that starts from a theory of error (how the program could be broken) and generates tests optimized for errors of that type.
  - Example: Boundary analysis (domain testing) is optimized for misclassification errors (IF A<5 miscoded as IF A<=5)
- Quicktesting may be more like scripted testing or more like ET
  - depends on the mindset of the tester.

To learn more about quicktests, see the risk-based testing keynote slides.

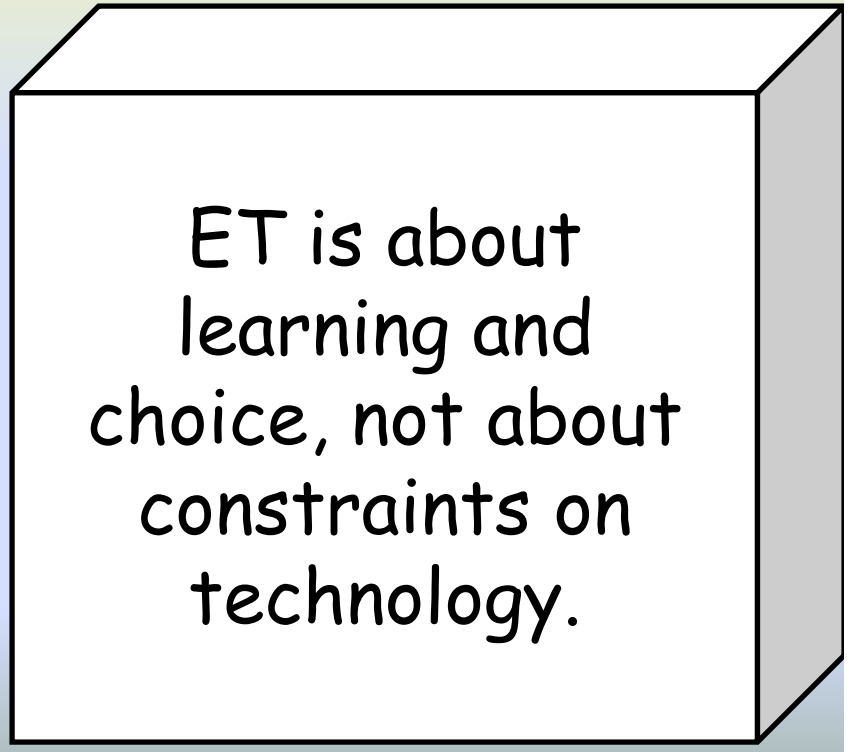
Areas of controversy

- ET is not quicktesting
- **ET is not only functional testing:**
  - Some programmers define testing narrowly
    - Agile ™ system testing focused around customer stories—not a good vehicle for parafunctional attributes
    - Parafunctional work is dismissed as peripheral
  - If quality is value to the stakeholder
    - and if value is driven by usability, security, performance, aesthetics, (etc.)
    - then testers should investigate these aspects of the product.



Areas of controversy

- ET is not quicktesting
- ET is not only functional testing
- **ET can involve tools of any kind and can be as computer-assisted as anything else we would call "automated"**
  - Along with
    - traditional "test automation" tools,
  - Emerging tool support for ET such as
    - Test Explorer
    - BBTest Assistant
  - and better thought support tools
    - Like Mind Manager and Inspiration
    - Qualitative analysis tools like Atlas.ti



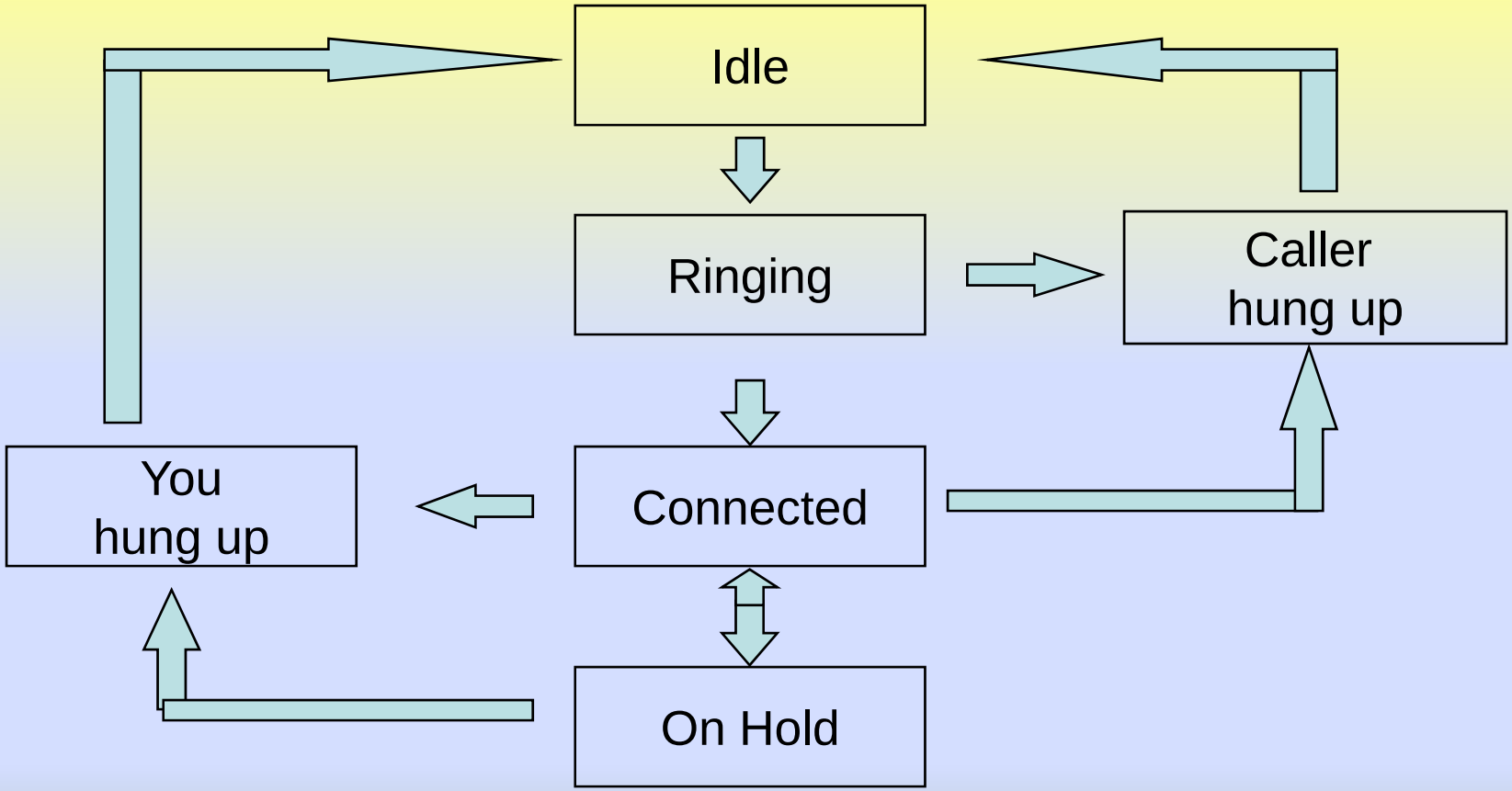
The Telenova stack failure

Telenova Station Set 1. Integrated voice and data.
108 voice features, 110 data features. 1984.

The Telenova stack failure

Context-sensitive display
10-deep hold queue
10-deep wait queue

The Telenova stack failure:
A simplified state diagram showing the bug



The underlying bug:

Beta customer (stock broker) had random failures

Could be frequent at peak times

An individual phone would crash and reboot. Others crashed while the first was rebooting

One busy day, service was disrupted all afternoon

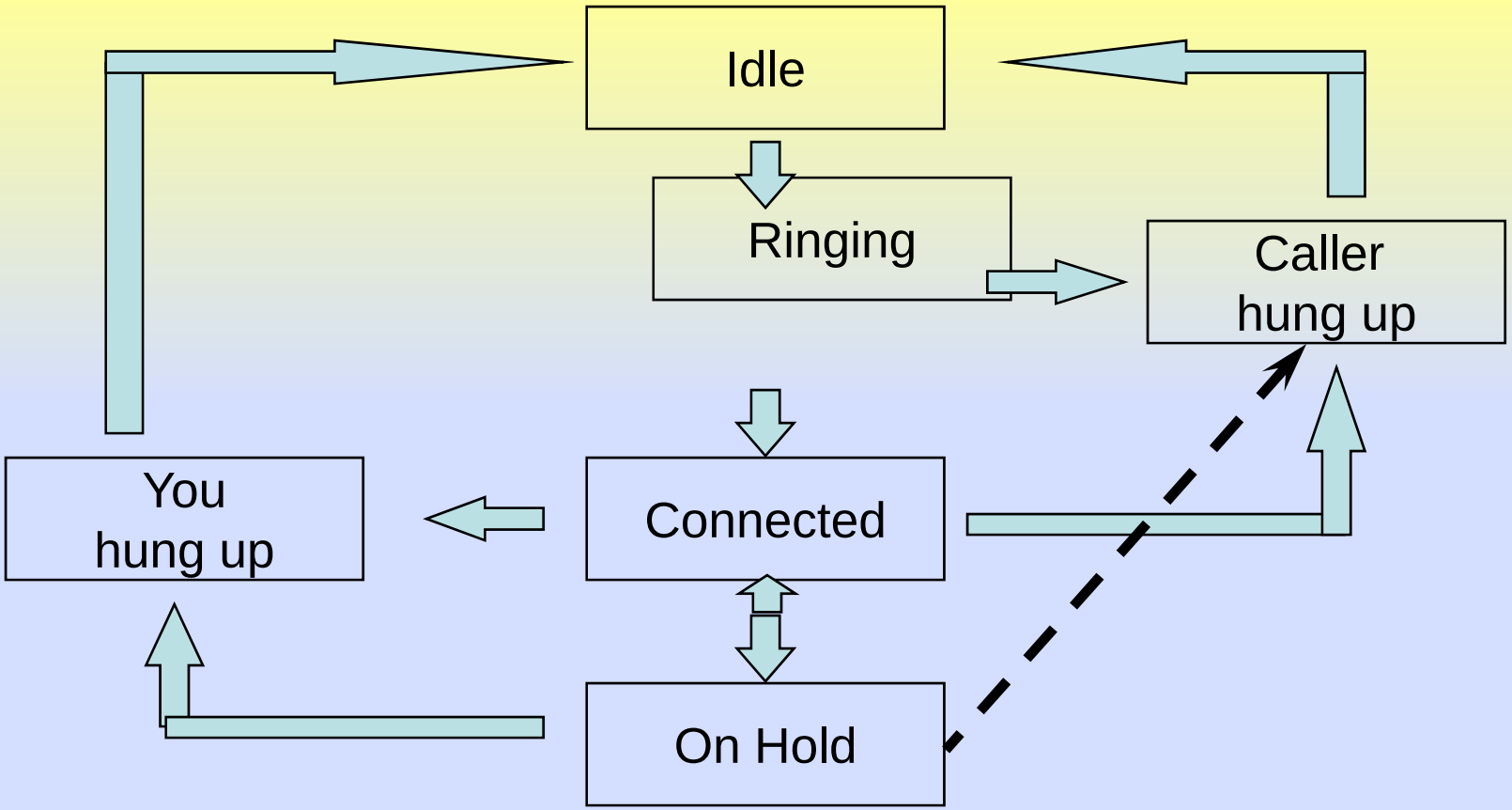
We were mystified:

- All individual functions worked
- We had tested all lines and branches.

Ultimately, we found the bug in the hold queue

- Up to 10 held calls, each adds record to the stack
- Initially, the system checked stack whenever it added or removed a call, but this took too much system time. We dropped the checks and added:
  - Stack has room for 20 calls (just in case)
  - Stack reset (forced empty) when we knew it should be empty
- Couldn't overflow the stack in the lab because we didn't know how to hold more than 10 calls.

The magic error



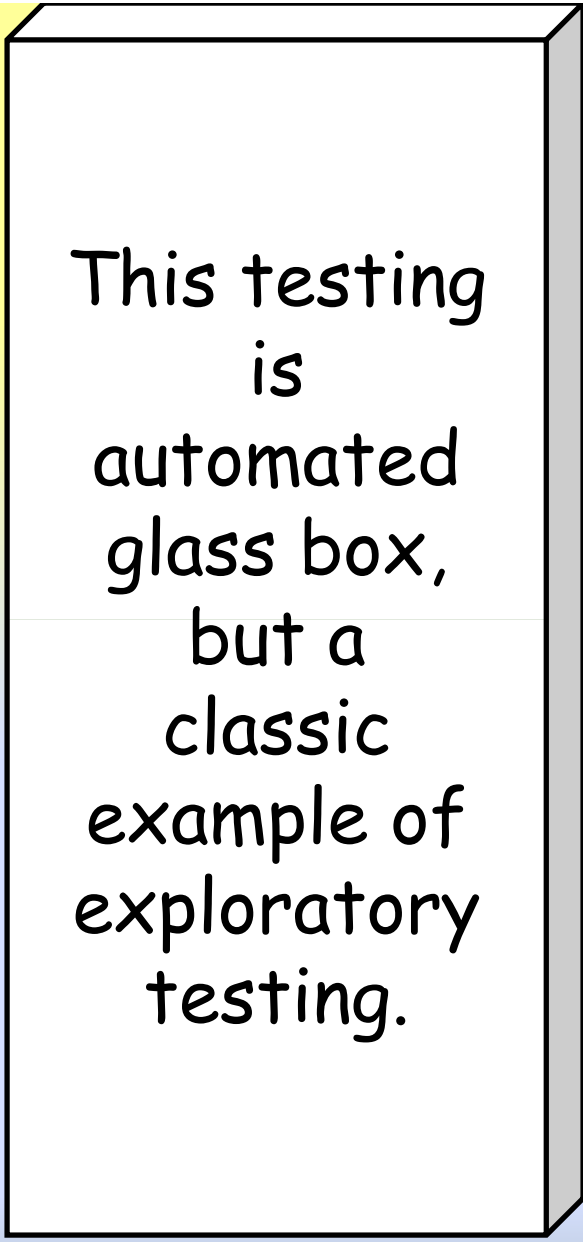
Telenova stack failure

Having found and fixed
the hold-stack bug,
should we assume
we've taken care of the problem
or that if there's one long-sequence bug,
there will be more?

Hmmm…
If you kill a cockroach in your kitchen,
do you assume
you've killed the last bug?
Or do you call the exterminator?

Simulator with probes

- Telenova (*) created a simulator
  - generated long chains of random events, emulating input to the system's 100 phones
  - could be biased, to generate more holds, more forwards, more conferences, etc.
- Programmers selectively added probes (non-crashing asserts that printed alerts to a log)
  - can't probe everything b/c of timing impact
- After each run, programmers and testers tried to replicate / fix anything that triggered a message
- When logs ran almost clean, shifted focus to next group of features.
- Exposed lots of bugs



(*) By the time this was implemented, I had joined Electronic Arts.

Areas of controversy

- ET is not quicktesting
- ET is not only functional testing
- ET can involve tools of any kind and can be as computer-assisted as anything else we would call "automated"
- **ET is not focused primarily around test execution**
  - I helped create this confusion by initially talking about ET as a test technique.

Controversy: ET is not a technique

In the 1980's and early 1990's, I distinguished between

- The evolutionary approach to software testing
- The exploratory testing technique(s), such as:
  - Guerilla raids
  - Taxonomy-based testing and auditing
  - Familiarization testing (e.g. user manual conformance tests)
  - Scenario tests

Controversy: ET is not a technique

1999 Los Altos Workshop on Software Testing #7 on Exploratory Testing

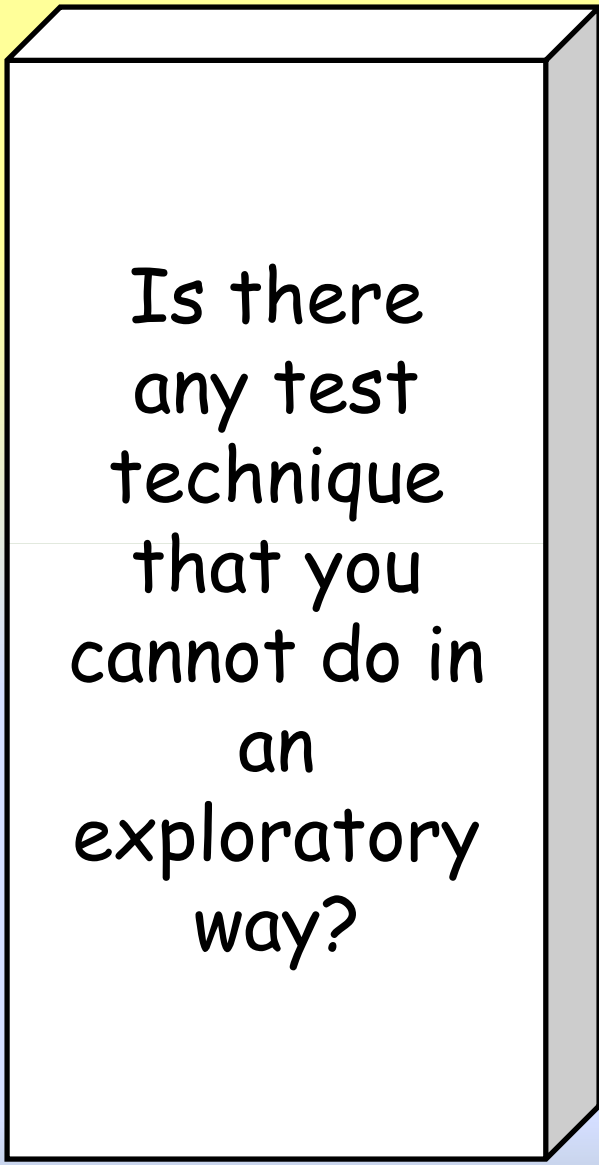
- James Tierney presented observations on MS "supertesters" indicating their strength is heavily correlated with social interactions in the development group (they translate what they learn from the team into tests)
- Bob Johnson and I presented a list of "styles of exploration" (a catalog of what we now call "quicktests")
- James Bach, Elisabeth Hendrickson, Harry Robinson, and Melora Svoboda gave presentations on models to drive exploratory test design

Controversy: ET is not a technique

At end of LAWST 7, David Gelperin concluded he didn't understand what is unique about "exploratory" testing. Our presentations all described approaches to design and execution of tests that he considered normal testing. What was the difference?

He had a point:

- Can you do domain testing in an exploratory way?
  - ***Of course***
- Specification-based testing?
  - ***Sure***
- Stress testing? Scenario testing? Model-based testing?
  - ***Yes, yes, yes***



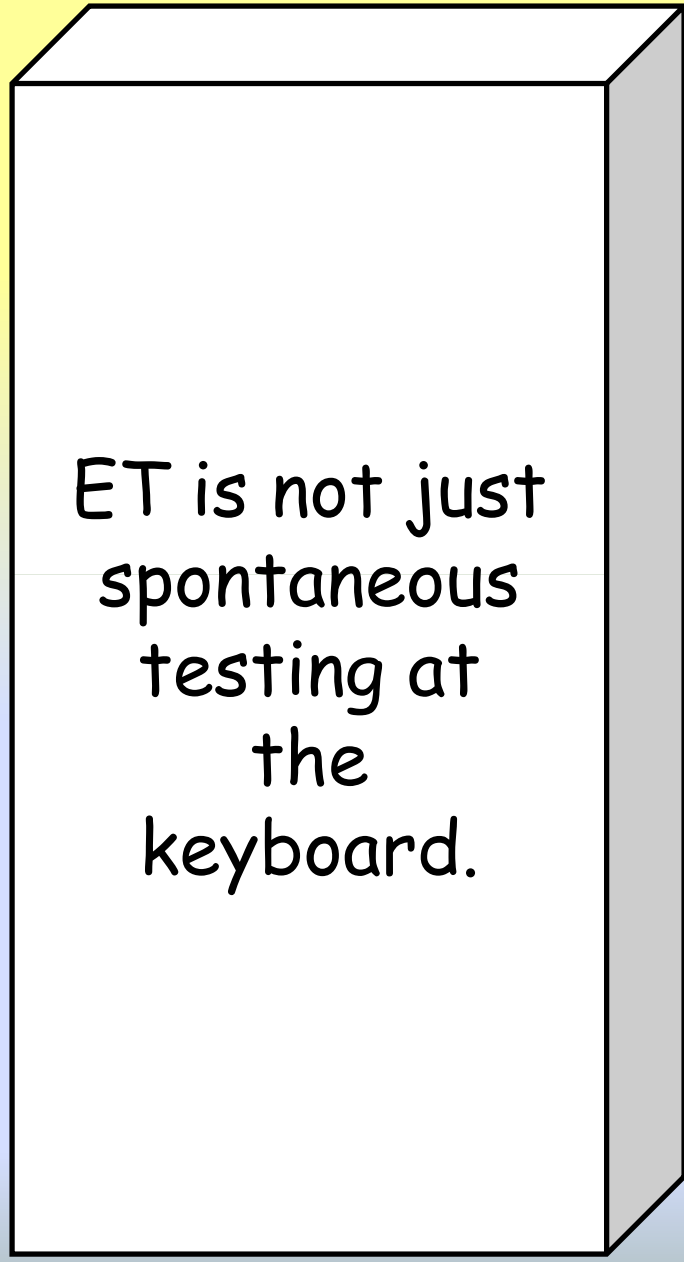
Controversy: ET is not a technique

WHET #1 and #2 – James Bach demonstrated that activities we undertake to learn about the product (in order to test it) are inherent in exploration.

- Of course they are
- But this became the death knell for the idea of ET as a technique
- **ET is a way of testing**
  - We learn about the product in its market and technological space (keep learning until the end of the project)
  - We take advantage of what we learn to design better tests and interpret results more sagely
  - We run the tests, shifting our focus as we learn more, and learn from the results.

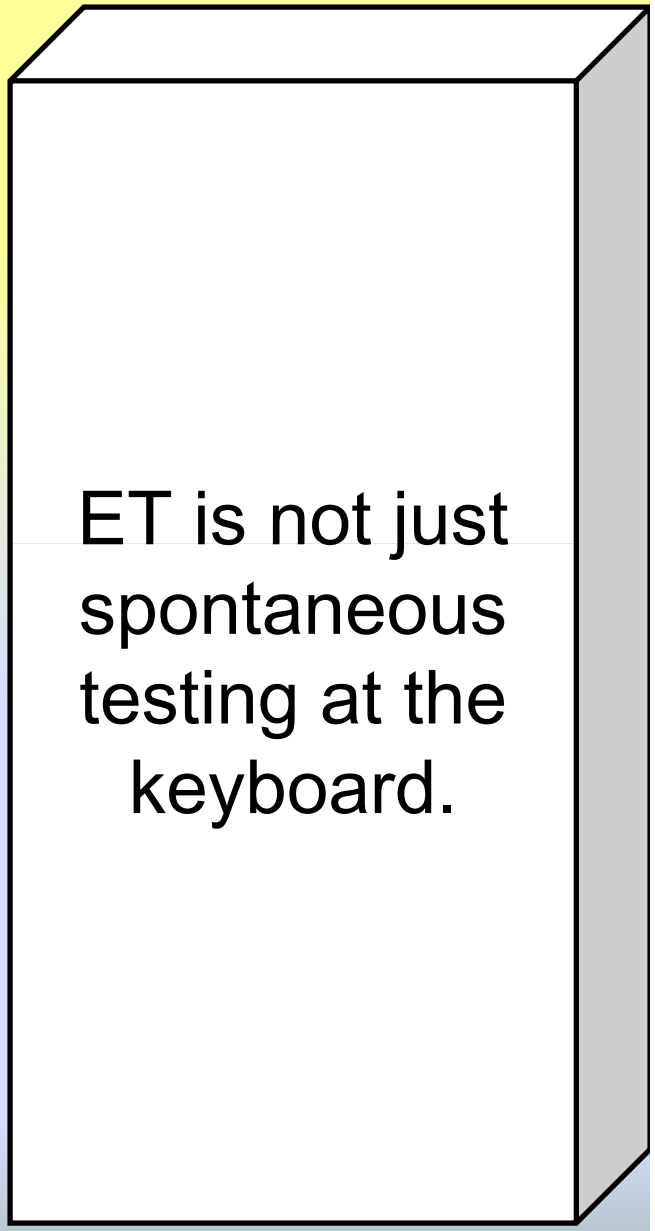
Areas of controversy

- ET is not quicktesting
- ET is not only functional testing
- ET can involve tools of any kind and can be as computer-assisted as anything else we would call "automated"
- ET is not focused primarily around test execution
- **ET can involve complex tests that require significant preparation**
  - Scenario testing is the classic example
  - To the extent that scenarios help us understand the design (and its value), we learn most of what we'll learn in the development and first execution. Why keep them?

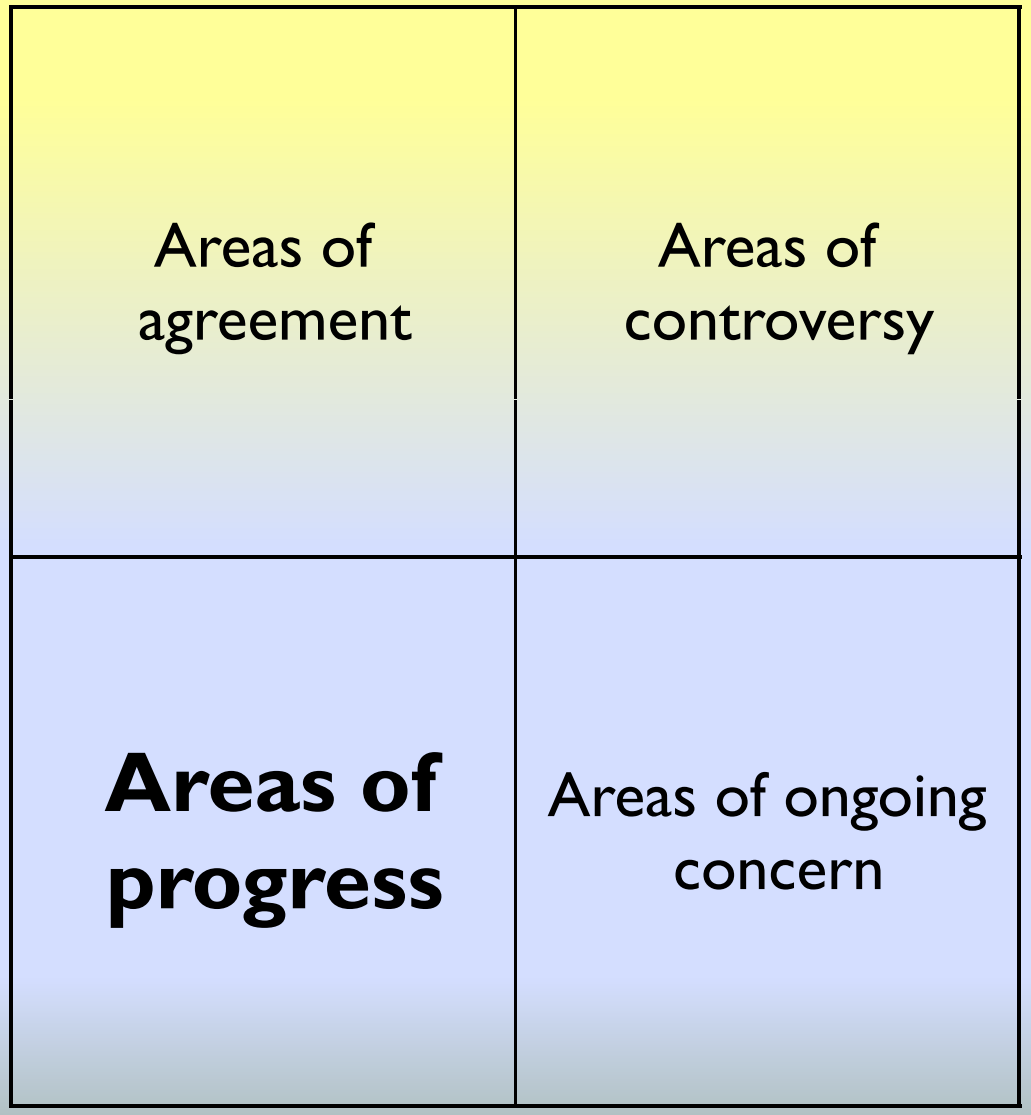


Areas of controversy

- ET is not quicktesting
- ET is not only functional testing
- ET can involve tools of any kind and can be as computer-assisted as anything else we would call "automated"
- ET is not focused primarily around test execution
- ET can involve complex tests that require significant preparation
- **ET is not exclusively black box**
  - "Experimental program analysis: A new paradigm for program analysis" by Joseph Ruthruff (Doctoral symposium presentation at International Conference on Software Engineering, 2006)



Exploratory testing after 24 years



Areas of progress

- We know a lot more about quicktests
  - Well documented examples from Whittaker's *How to Break…* series and Hendrickson's and Bach's courses

Areas of progress

- We know a lot more about quicktests
- We have a better understanding of the oracle problem and oracle heuristics

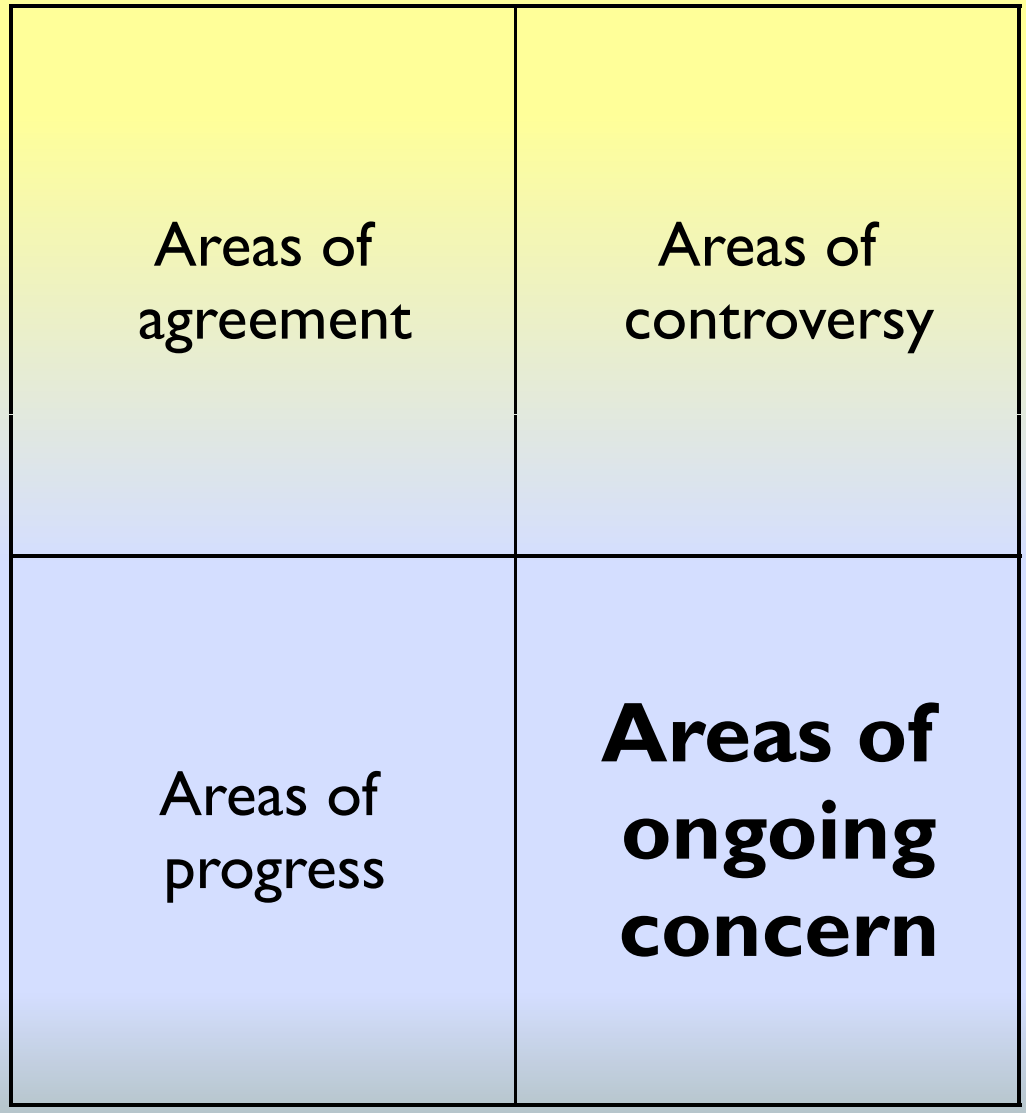
Areas of progress

- We know a lot more about quicktests
- We have a better understanding of the oracle problem and oracle heuristics
- We have growing understanding of ET in terms of theories of learning and cognition

Areas of progress

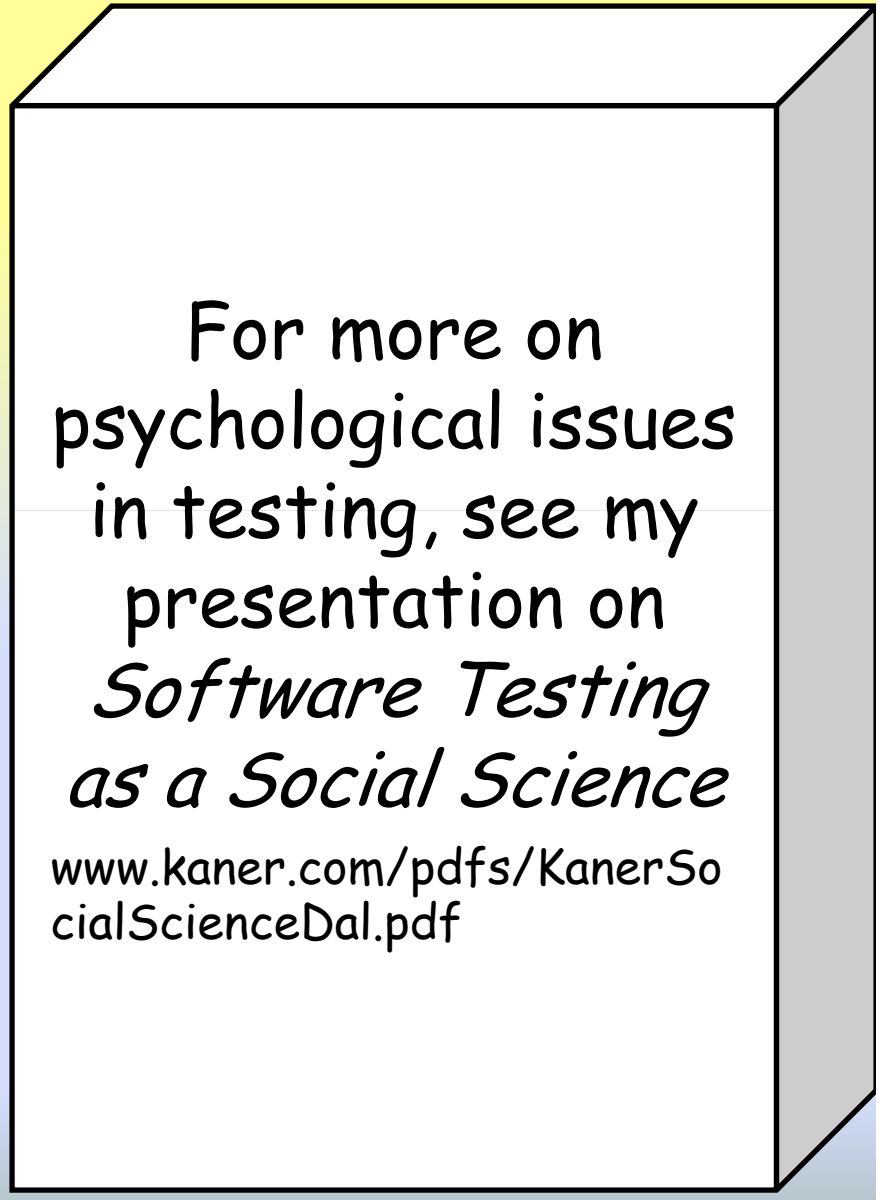
- We know a lot more about quicktests
- We have a better understanding of the oracle problem and oracle heuristics
- We have growing understanding of ET in terms of theories of learning and cognition
- **We have several guiding models**
  - We now understand that models are implicit in all tests
  - Failure mode & effects analysis applied to bug catalogs
  - Bach / Bach / Kelly's activities model
  - Satisfice heuristic test strategy model
  - State models
  - Other ET-supporting models (see Hendrickson, Bach)

Exploratory testing after 24 years



Areas of ongoing concern

- Testing is
  - more skilled and cognitively challenging
  - more fundamentally multidisciplinary
  - than popular myths expect



Areas of ongoing concern

- Testing is more skilled and cognitively challenging, more fundamentally multidisciplinary, than popular myths expect:
- Unskilled testing shows up more starkly with ET

Areas of ongoing concern

Testing is more skilled and cognitively challenging, more fundamentally multidisciplinary, than popular myths expect:

Unskilled testing shows up more starkly with ET

- Repetition without realizing it
- Areas missed without intent
- Incorrect perception of depth or coverage
- Tester locks down on a style of testing without realizing it
- Wasted time due to reinvention of same tests instead of reuse
- Wasted effort creating test data
- Audit fails because of lack of traceability
- Weak testing because the tester is unskilled and tests are unreviewed
- Difficult to document the details of what was done
- May be difficult to replicate a failure
- Hard to coordinate across testers
- Harder to spot a failure.

The essence of ET is learning (and learning about learning)

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

The individual contributor (tester rather than “test planner” or manager)

Areas of ongoing concern

- Testing is more skilled and cognitively challenging, and more fundamentally multidisciplinary, than popular myths expect
- **What level of skill, domain knowledge, intelligence, testing experience (overall "strength" in testing) does exploratory testing require?**
  - We are still early in our wrestling with modeling and implicit models
    - How to teach the models
    - How to teach how to model

The essence of ET is learning (scripted execution)

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

The individual contributor (tester rather than "test planner" or manager)

The essence of ET is learning (exploratory execution)

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

The individual contributor (tester rather than "test planner" or manager)

The essence of ET is learning (learning to explore)

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

The individual contributor (tester rather than "test planner" or manager)

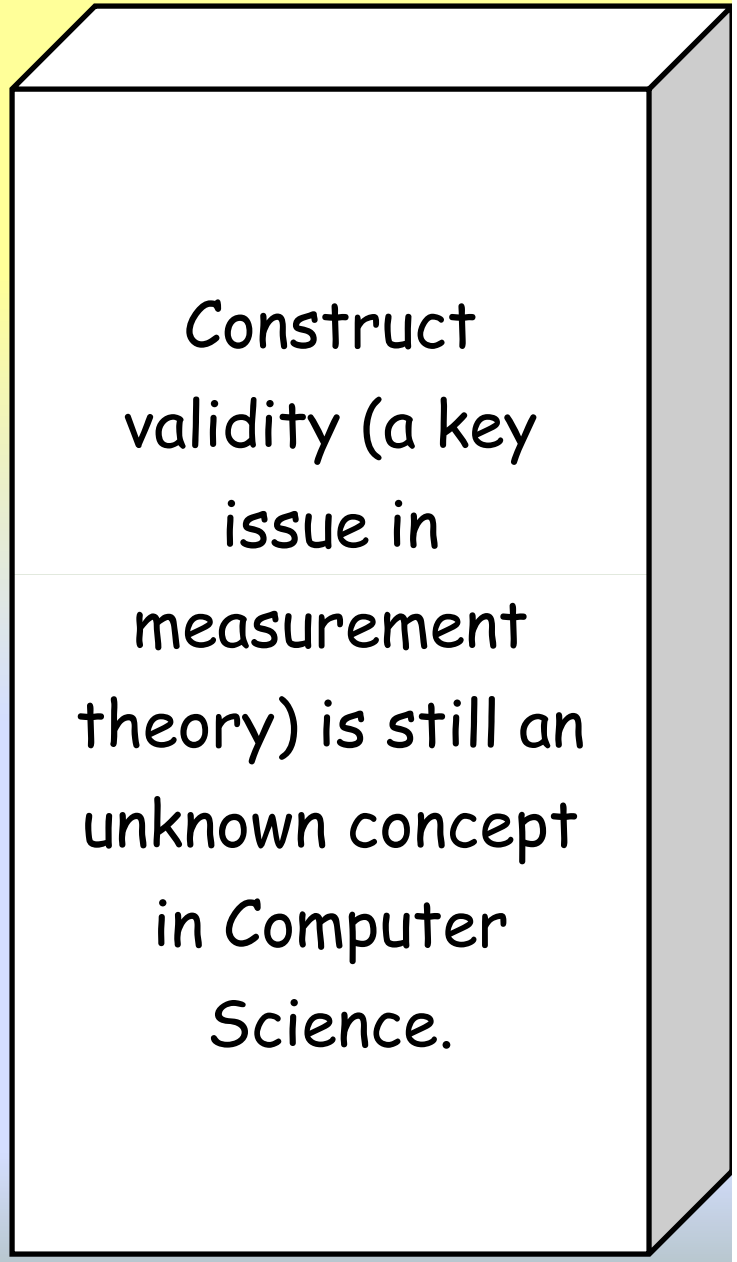
The essence of ET is learning (and learning about learning)

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

The individual contributor (tester rather than "test planner" or manager)

Areas of ongoing concern

- Testing is more skilled and cognitively challenging, and more fundamentally multidisciplinary, than popular myths expect
- What level of skill, domain knowledge, intelligence, testing experience (overall "strength" in testing) does exploratory testing require?
- **We are just learning how to assess individual tester performance**



Areas of ongoing concern

- Testing is more skilled and cognitively challenging, and more fundamentally multidisciplinary, than popular myths expect
- What level of skill, domain knowledge, intelligence, testing experience (overall "strength" in testing) does exploratory testing require?
- We are just learning how to assess individual tester performance
- **We are just learning how to track and report status**
  - Session based testing
  - Workflow breakdowns
  - Dashboards

Areas of ongoing concern

- Testing is more skilled and cognitively challenging, and more fundamentally multidisciplinary, than popular myths expect
- What level of skill, domain knowledge, intelligence, testing experience (overall "strength" in testing) does exploratory testing require?
- We are just learning how to assess individual tester performance
- We are just learning how to track and report status
- **We don't yet have a good standard tool suite**
  - Tools guide thinking
  - Hendrickson, Bach, others have made lots of suggestions

Closing notes

- If you want to attack any approach to testing as unskilled, attack scripted testing
- If you want to hammer any testing approach on coverage, look at the fools who think they have tested a spec or requirements document when they have one test case per spec item, or code with one test per statement / branch / basis path.
- Testing is a skilled, fundamentally multidisciplinary area of work.
- Exploratory testing brings to the fore the need to adapt to the changing project with the information available.