The Pattern | Wordpad

Rapid Software Testing™

V3.2.2

James Bach, Satisfice, Inc.
james@satisfice.com
www.satisfice.com
+1 (360) 440-1435

Michael Bolton, DevelopSense
mb@developsense.com
www.developsense.com
+1 (416) 656-5160

Acknowledgements

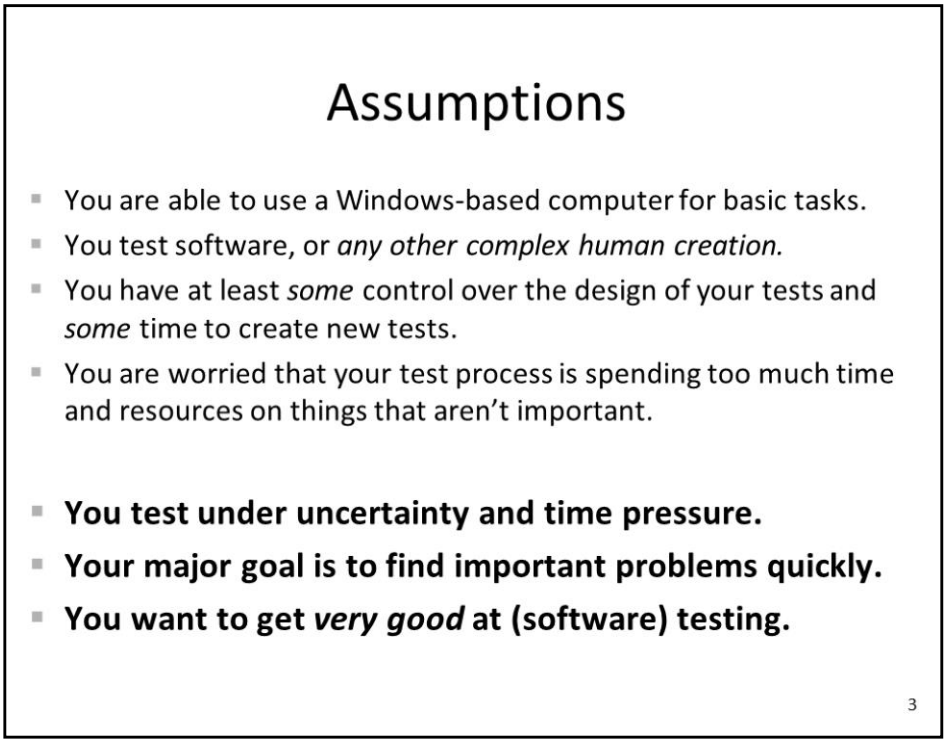
- **James Bach, Michael Bolton,** and **Paul Holland** co-create and teach this class.
- Some of this material was developed in collaboration with **Cem Kaner**.
- **Jon Bach** has been a long-term collaborator in developing ET management and training methods.
- Many of the ideas in this presentation were also inspired by or augmented by other colleagues including Doug Hoffman, Bret Pettichord, Brian Marick, Dave Gelperin, Elisabeth Hendrickson, Jerry Weinberg, Noel Nyman, and Mary Alton.
- Some of our exercises were introduced to us by Payson Hall, Ross Collard, James Lyndsay, Dave Smith, Earl Everett, Brian Marick, and Joe McMahon.
- Many ideas were improved by students who took earlier versions of the class going back to 1995.

2

Although the value of the class is greater because of the people who helped us, the sole *responsibility* for the content of this class belongs to the authors, James Bach and Michael Bolton. This class is essentially a long editorial opinion about testing. We have no authority to say what is right. There are no authorities in our field. We don't peddle best practices. We don't believe in them.

What we will do is share our experiences and challenge your mind. We hope you will challenge us back. You'll find the class more rewarding, we think, if you assume that our purpose is to make you stronger, smarter and more confident as a tester. Our purpose is NOT to have you listen to what the instructor says and believe it, but to listen and then *think for yourself*.

All good testers think for themselves. In a way, *that's what testing is about*. We look at things differently than everyone else so that we can find problems no one else will find.



If any of these assumptions don't fit, this class might not be for you.

We don't make any assumptions about how experienced you are. The class can work well for novices or experts.

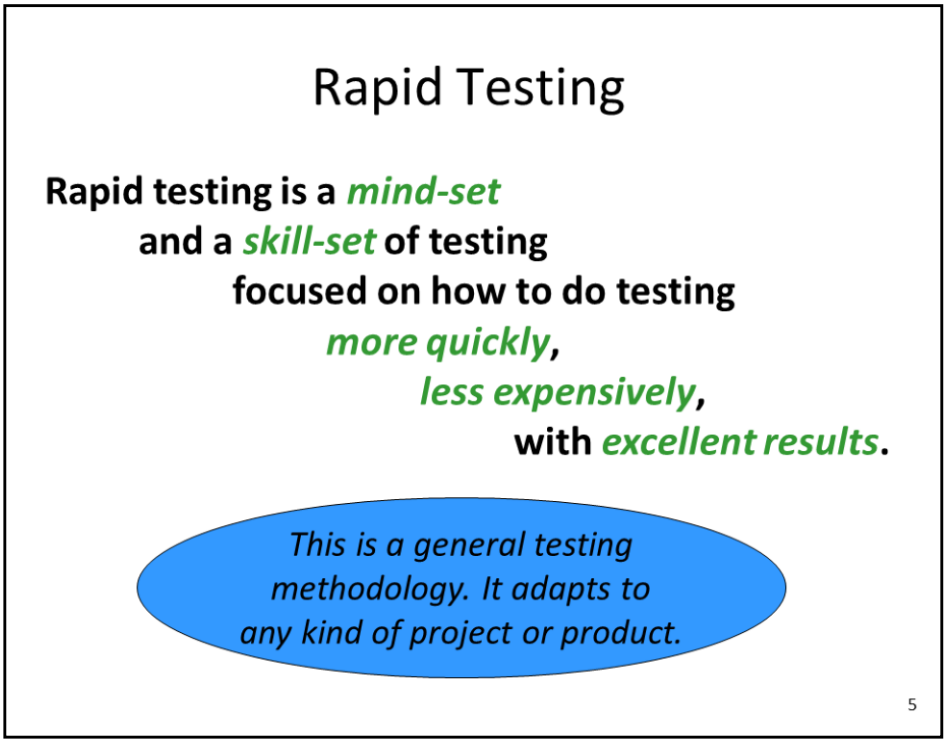
It's the desire to be very good at software testing that compels you to *want* to take a class. Most of the people in the world who work in software testing have never taken a testing class, read a testing book, attended a testing conference, participated in a test user group. They still get to keep their jobs, and they might even be okay at testing. A class like this becomes important when you want to be confident and great at software testing.

As a supplement to this course, consider the Black Box Software Testing course available through http://www.testingeducation.org. This course, developed by Cem Kaner and James Bach, is set up for self-study. It's composed of more than 40 hours of video lectures, and includes detailed course notes, quizzes, self-tests, supplementary reading materials, and links to more. The Association for Software Testing (http://associationforsoftwaretesting.org) offers instructor-led versions of the course to its members.

Non-Assumptions

- You **do not** have to be an actual working tester. Perhaps you work with testers or want to learn about how a good tester operates.
- You **do not** have to be coder, but you *can* approach this material *as* a coder.
- It **doesn't matter** what you test. Hardware, software, anything…
- It **doesn't matter** what development process you use. Agile, Fragile, Crash Dummy, Bungee Jump, Rollerball, Moneyball, Ballroom dance, River Dance, or Waterfall… This is a mental discipline which applies to anything.

- **If you need to FAKE a testing project, this class will not help.**

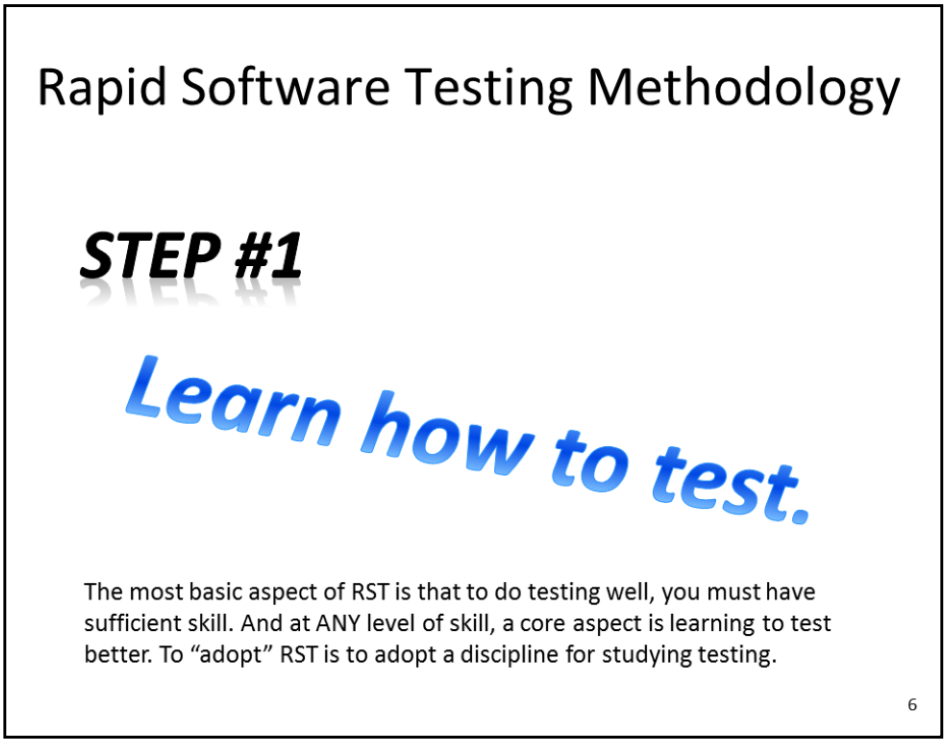


Rapid testing is a mind set–a particular way of looking at the world; and a skill set—a particular set of things that we practice and get better at.

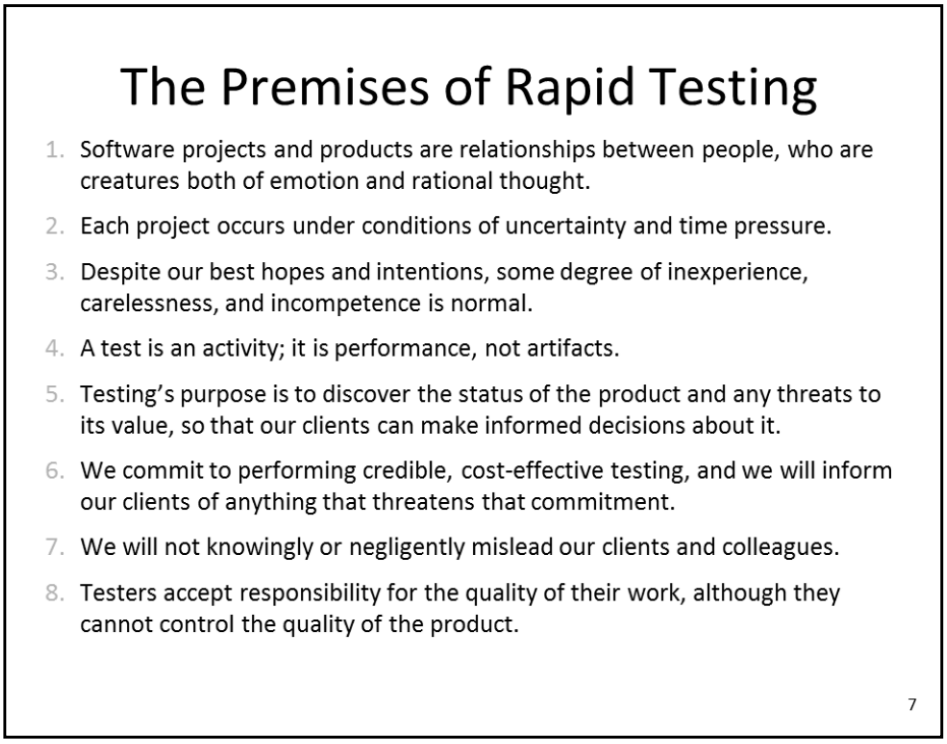
We're advocates of the context-driven approach to software testing.

We've applied these approaches in financial institutions, in court cases, in testing of medical devices, in commercial shrink-wrapped software, to games… The methodology is designed to adapt to any kind of testing context.

Rapid testing involves considerations of skill, personal integrity, improved focus on the underlying need for testing tasks, improved appreciation for the stakeholders of testing tasks, and knowledge of the possible techniques and tools that could be brought to bear to improve efficiency.



A methodology is a "system of methods." It's a "how-to." But in testing a lot of the how-to is tacit (unspoken). To learn to test you must experience testing and struggle to solve testing problems. The explicit (written, spoken, or pictured) part of methodology sits on top of that.



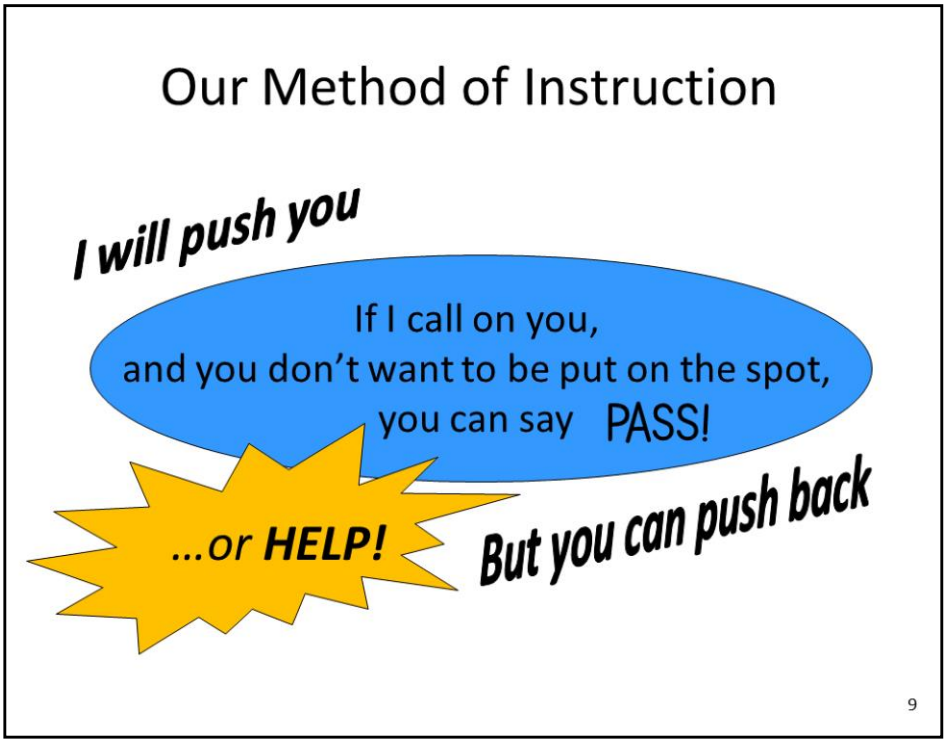
These are the premises of the Rapid Software Testing methodology. Everything in the methodology derives in some way from this foundation. These premises derive from our experience, study, and discussions over a period of decades. They have been shaped by the influence of two thinkers above all: Cem Kaner and Jerry Weinberg, both of whom have worked as programmers, managers, social scientists, authors, teachers, and of course, testers.

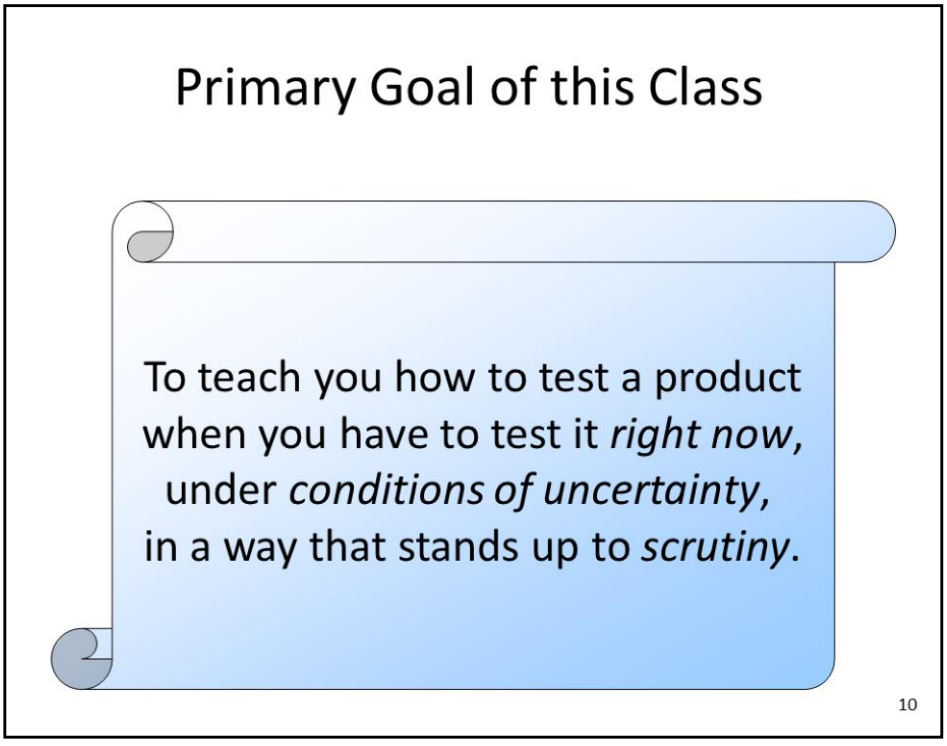
.

Our Method of Instruction

- **The Exercises are the Most Important Part:** We use *immersive socratic exercises* that are designed to fool you if you don't ask questions. We rarely provide all the information you need. *Asking questions is a fundamental testing skill!*
- **We Use the Socratic Method:** We will probably not ask you any question that has a simple or single good answer.
- **The Class Presents Our Editorial Opinions:** We do not make appeals to authority; we speak only from our experiences, and we appeal to your experience and intelligence.
- **Not All Slides Will be Discussed:** There is *much* more material here than we can cover in detail, so we may skip some of it. (If you want me to return to something that I skipped, just ask.)
- **We Need to Hear from You:** You control what you think and do, so we encourage you to *question* and *challenge* the lecture. (Talk to me during the break, too.)

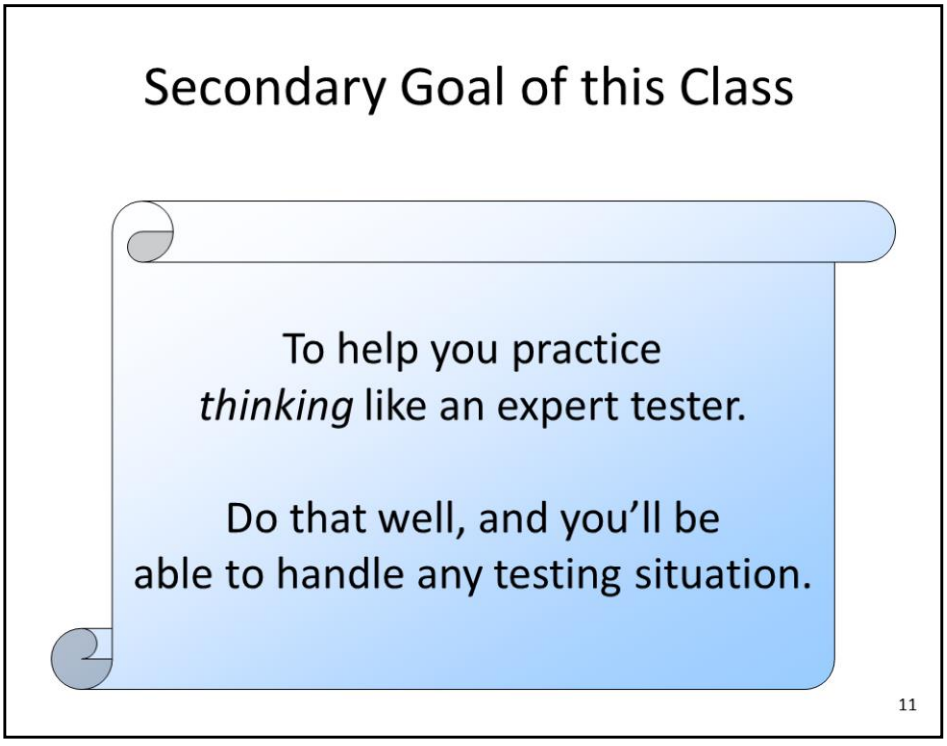
8





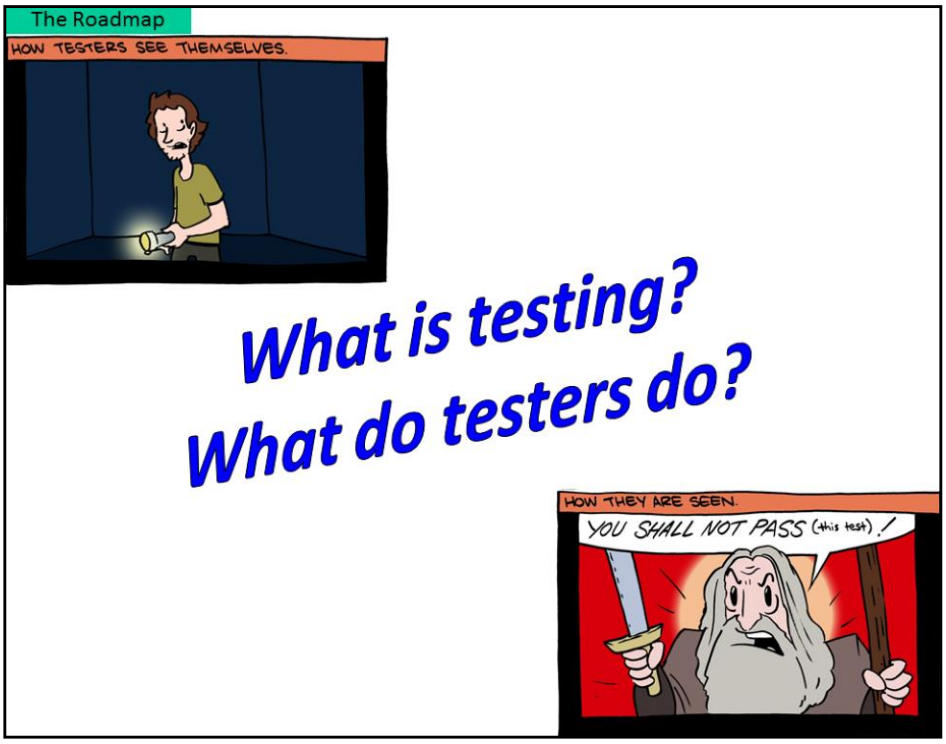
Without scrutiny, this would be easy to do. You could test badly, but no one would ever know or care. What makes testing hard is that someone, perhaps your employer, perhaps the users, or perhaps you yourself, will be judging your work. Maybe this scrutiny will be indirect, such as someone unhappy with the product and cursing your company for building it so poorly. Or perhaps it will be the most intimate form of scrutiny– which is your feeling of pride or disappointment in your own work.

If you want your testing to pass muster, you need to work at it.



You will get some practice in thinking like an expert tester. That will mean some success and some failures. For the successes, we want you to feel good—strong and powerful; that's a feeling that an expert tester gets. Alas, from time to time, expert testers also get the opportunity to feel embarrassed by failure. There are many traps in the exercises. We want to give you the experience of being trapped, of recognizing the traps, and of working your way out of them. This class is intended to provide you with a safe place to do that, so you can recognize similar situations and respond appropriately when you're on the job.

We also *expect* you to challenge our expertise. That's part of developing your own expertise, but it's also fundamental to our development. Any "expert" can be wrong, and "expert advice" is always wrong in some context, so we don't get upset if you argue. On the contrary; we're delighted if you argue. Challenging our statements shows us that you're thinking about how things might work—or not—in your own environment. The conversation is often very enlightening.

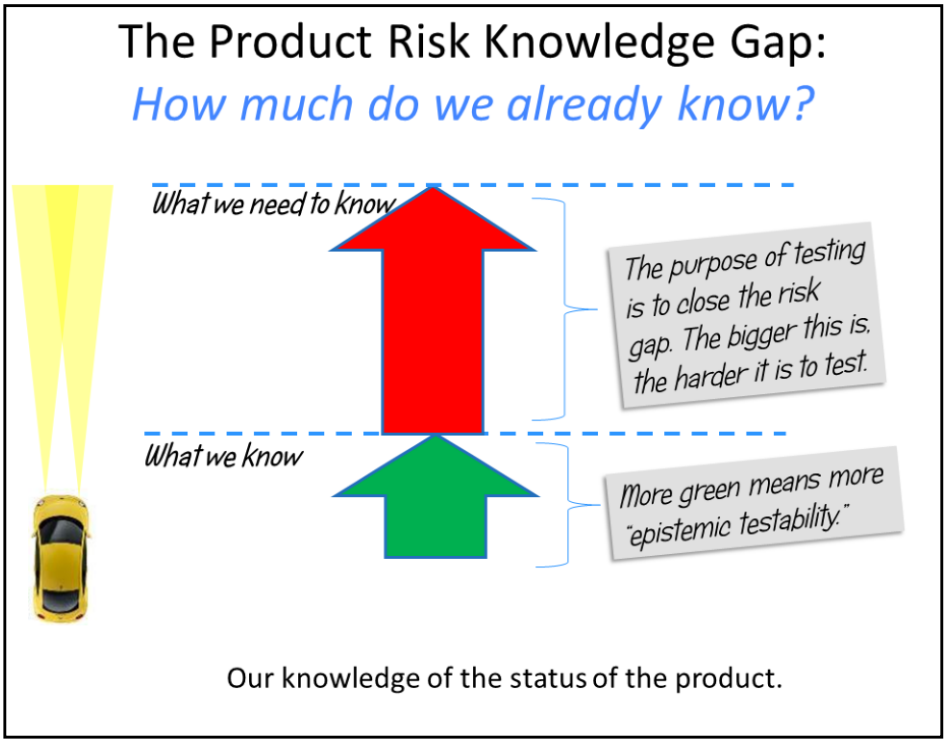


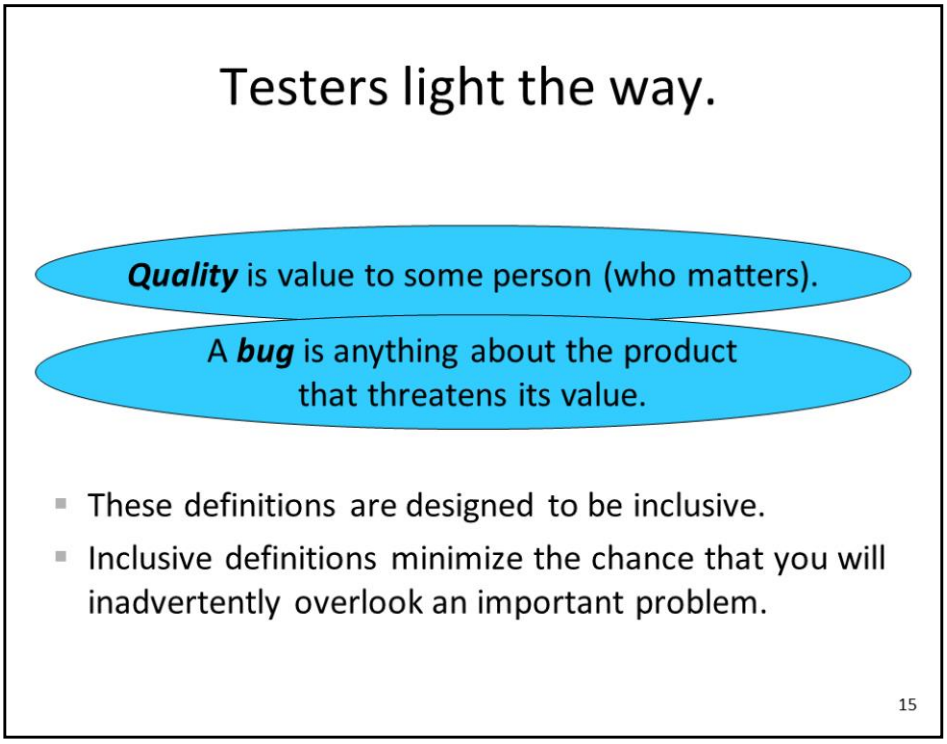
The Bug

Testers light the way.

This is our role.

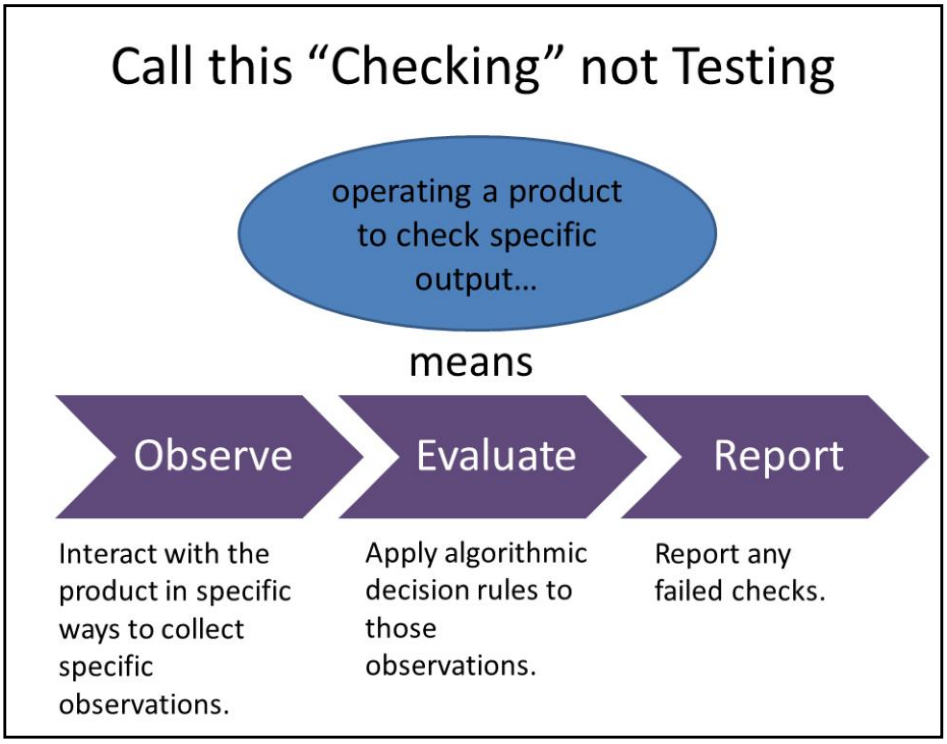
We see things for what they are.
We make informed decisions about quality possible,
because we think critically about software.

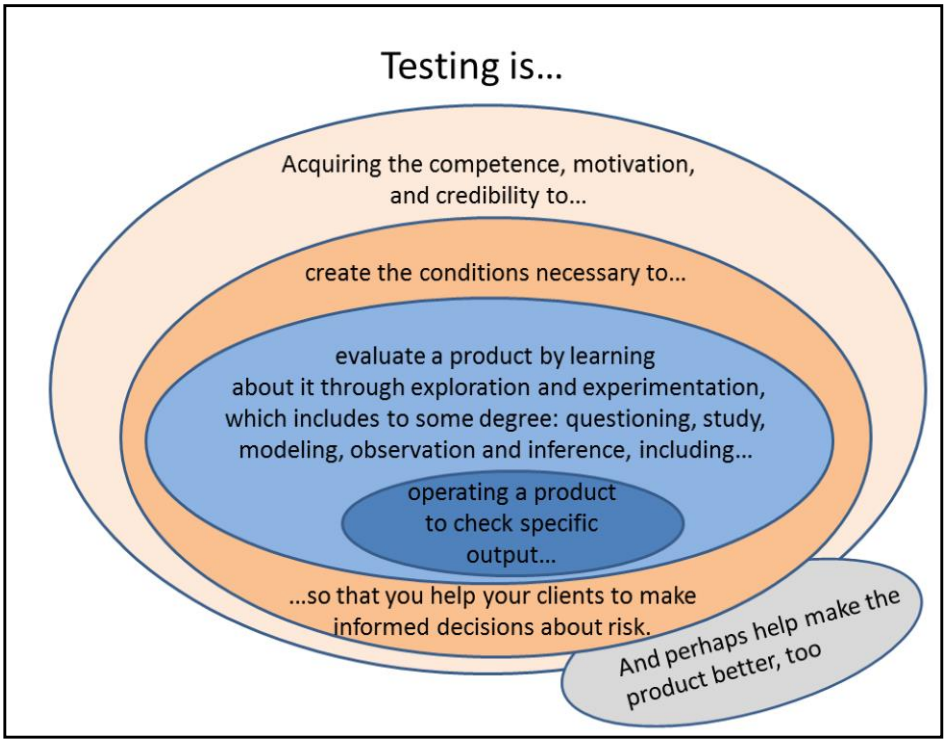


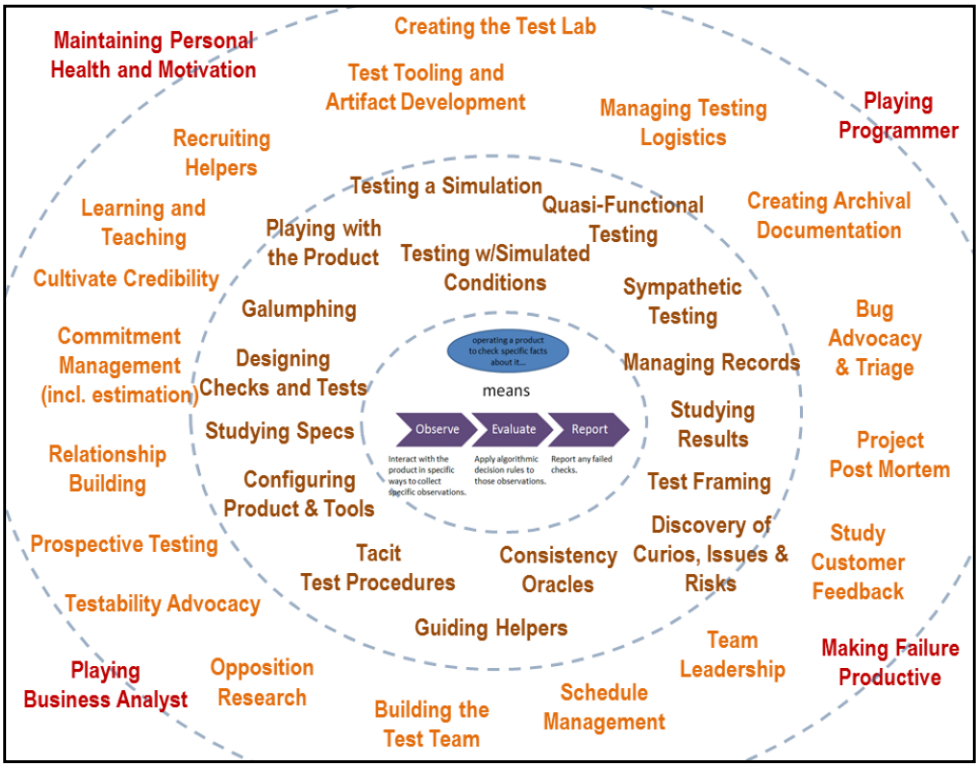


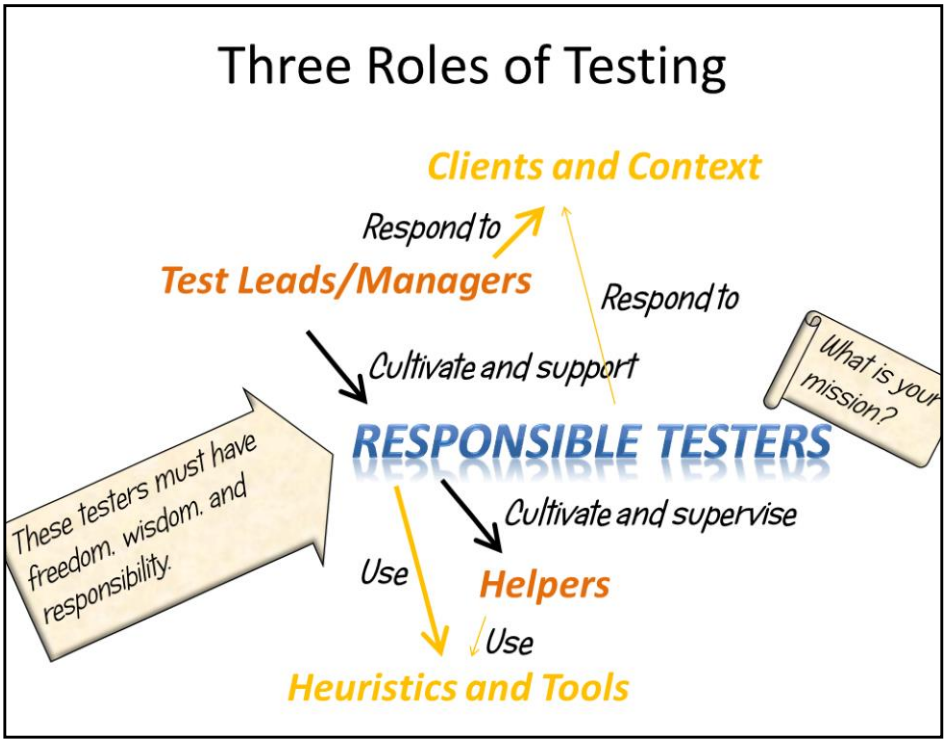
The first question about quality is always "whose opinions matter?" If someone who doesn't matter thinks your product is terrible, you don't necessarily change anything. So, an important bug is something about a product that really *bugs* someone *important*.

One implication of this is that, if we see something that we think is a bug and we're overruled, *we don't matter*. That's a fact: testers don't matter. That is to say, our standards are not the standards to which the product must adhere; we don't get to make decisions about quality. This can be hard for some testers to accept, but it's the truth. We provide information to managers; they get to make the hard decisions. They get the big bucks, and we get to sleep at night.





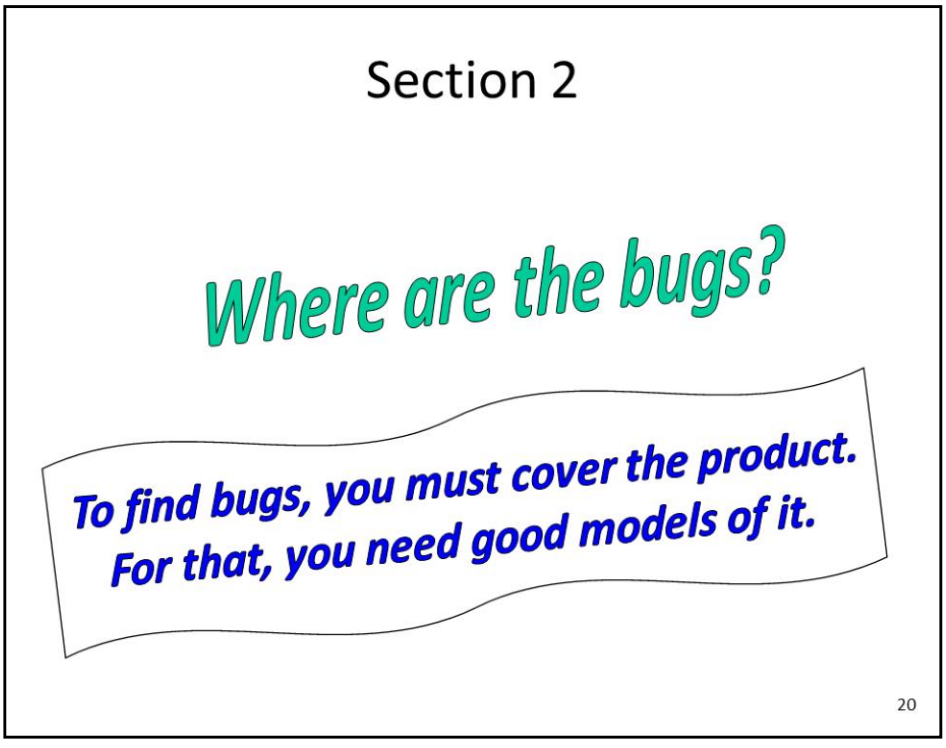


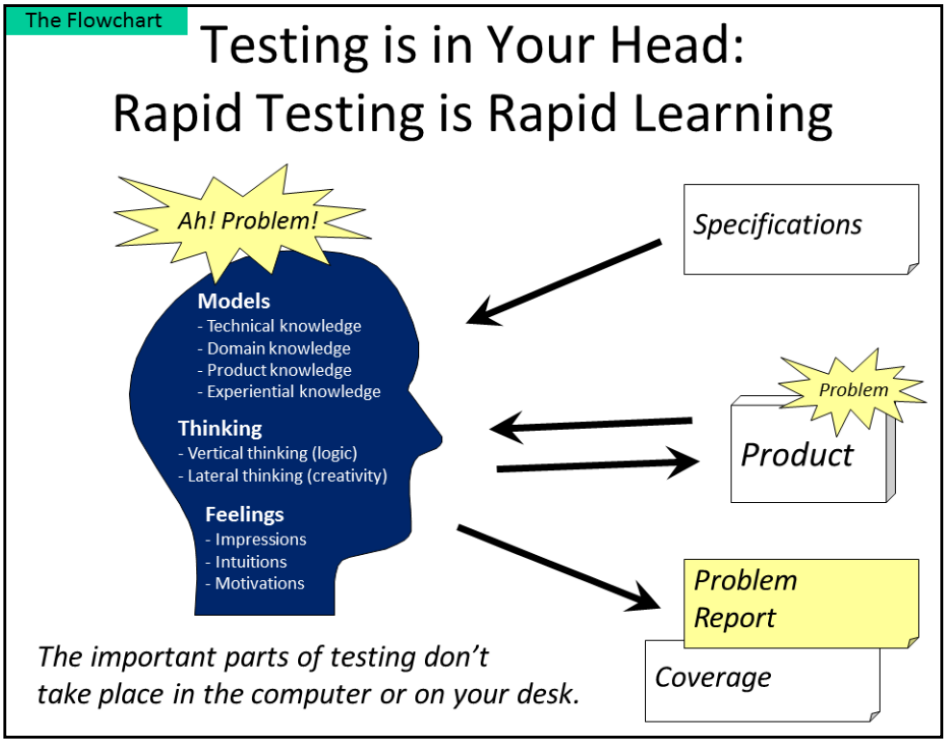


Freedom: The ability to do things that might hurt people (including yourself).

Wisdom: Knowing how not to hurt people (including yourself).

Responsibility: Choosing not hurt people (including yourself).




The Flowchart
Testing is in Your Head:
Rapid Testing is Rapid Learning
Ah! Problem!
Models
- Technical knowledge
- Domain knowledge
- Product knowledge
- Experiential knowledge
Thinking
- Vertical thinking (logic)
- Lateral thinking (creativity)
Feelings
- Impressions
- Intuitions
- Motivations
Specifications
Problem
Product
Problem Report
Coverage
The important parts of testing don't take place in the computer or on your desk.

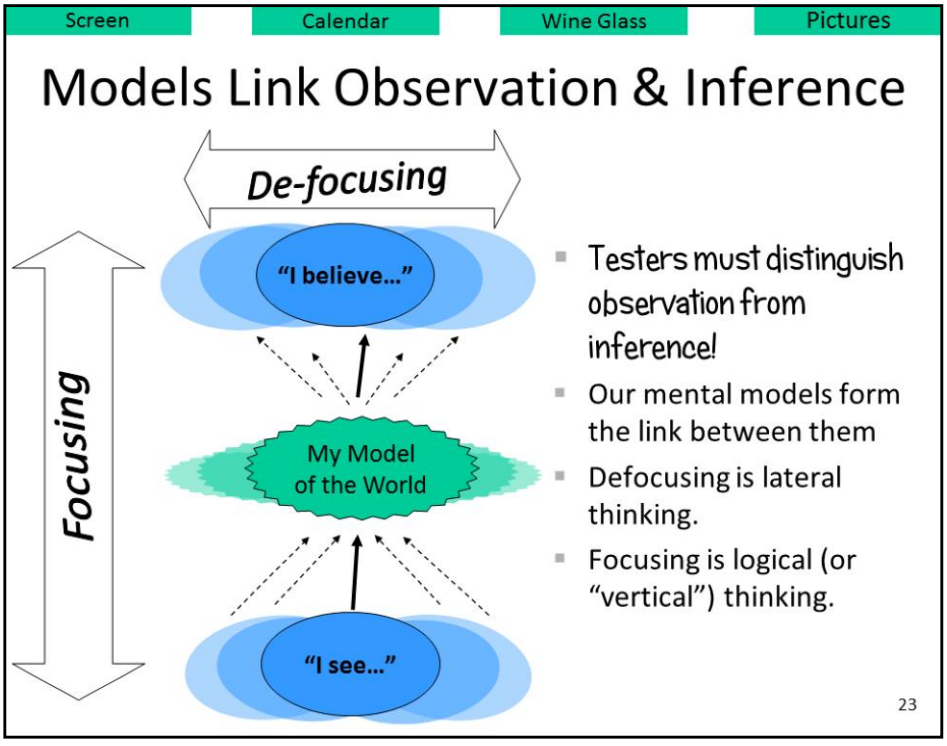
Models Link Observation and Inference

- **A model is an idea, activity, or object…**

  such as an idea in your mind, a diagram, a list of words, a spreadsheet, a person, a toy, an equation, a demonstration, or a program…
- **…that represents another idea, activity, or object.**

  such as something complex that you need to work with or study.
- **…A GOOD model is one that helps you understand or manipulate the thing that it represents.**
  - A map helps navigate across a terrain.
  - "2 + 2 = 4" is a model for adding two apples to a basket that already has two apples in it.
  - Atmospheric models help predict where hurricanes will go.
  - A fashion model helps people to understand how clothing would look on actual humans.
  - Your beliefs about what you test are a model of what you test.

22

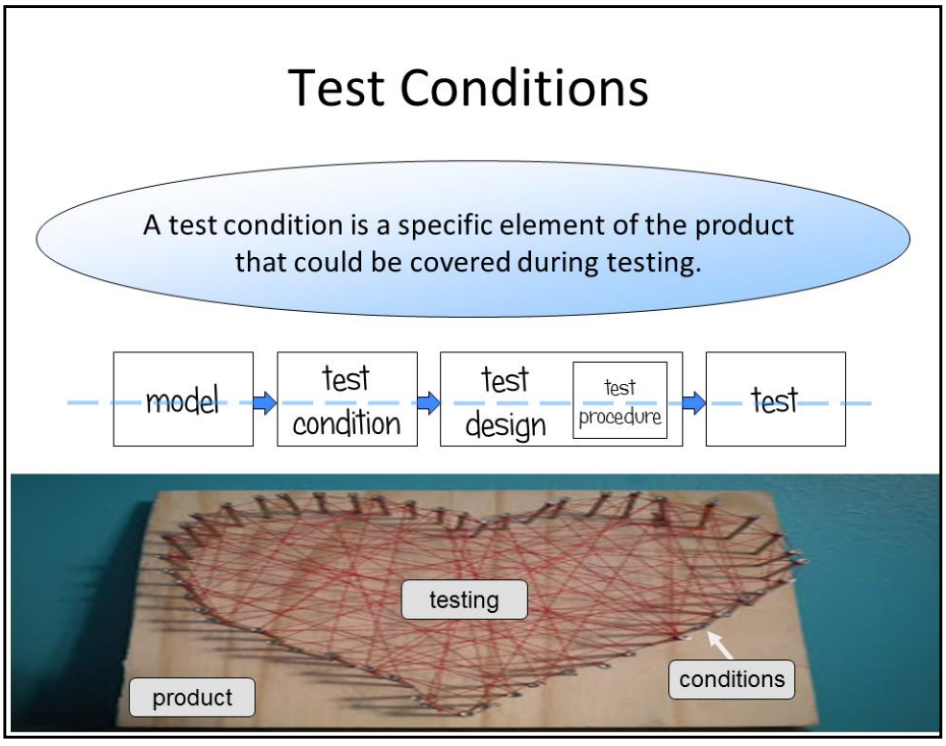


Datacenter

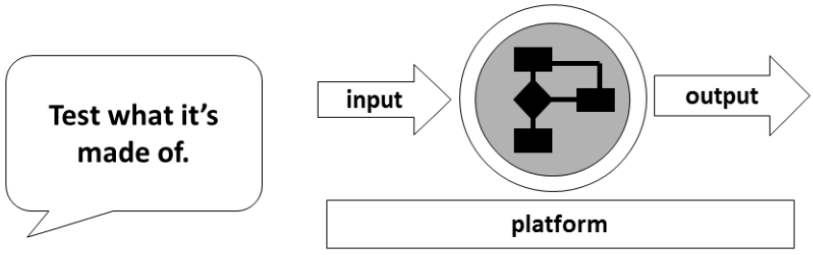
Coverage

____ **coverage** is how thoroughly you have examined the product with respect to some model of ____.

- **Interesting kinds of coverage**
  - Product coverage: *What aspects of the product did you look at?*
  - Risk coverage: *What risks have you tested for?*
  - Requirements coverage: *What requirements have you tested for?*

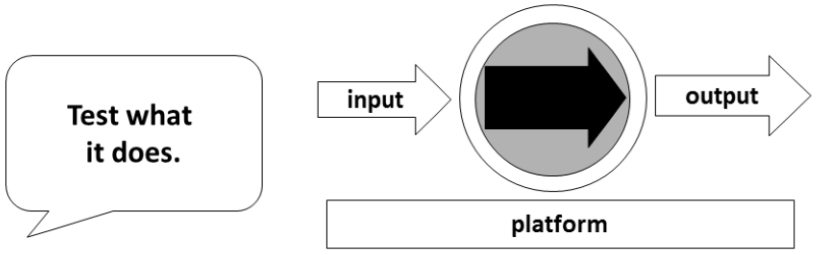


Product Elements:
Structural Coverage



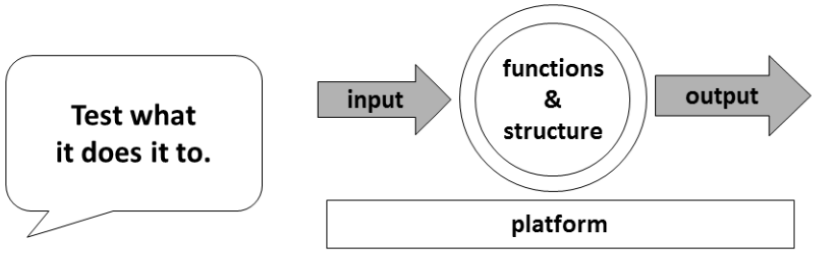
- Print testing example
  - Files associated with printing
  - Code modules that implement printing
  - Code statements inside the modules
  - Code branches inside the modules

Product Elements:
Functional Coverage



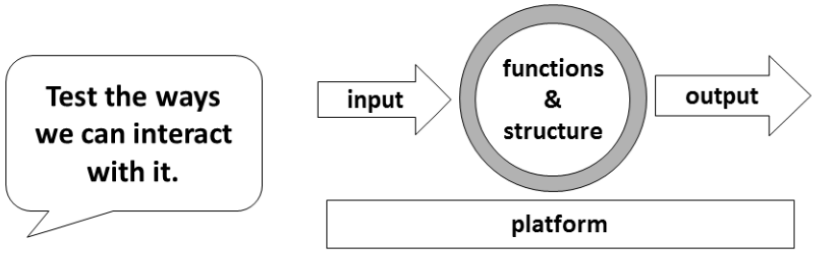
- Print testing example
  - Print, page setup and print preview
  - Print range, print copies, zoom
  - Print all, current page, or specific range

Product Elements:
Data Coverage



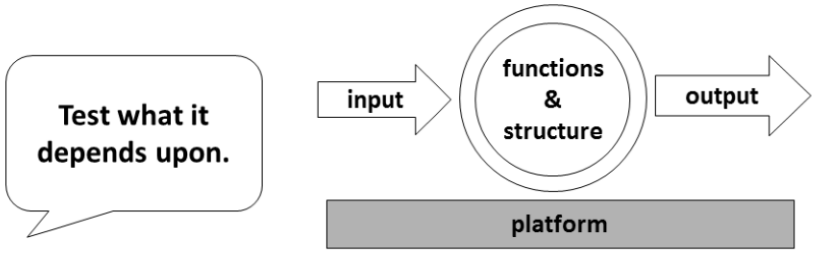
- Print testing example
  - Types of documents
  - Items in documents, size and structure of documents
  - Data about how to print (e.g. zoom factor, no. of copies)

Product Elements:
Interface Coverage



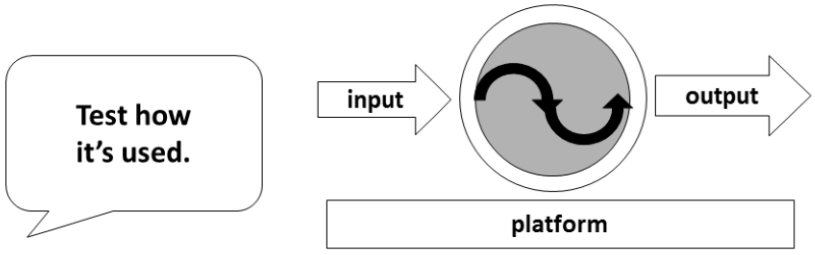
- Print testing example
  - User interface elements; menus, windows, dialogs, controls
  - Programming interfaces for launching and controlling a print job
  - Printing to or from files
  - USB printer interface protocol
  - Physical connections between printer and computers and/or network

Product Elements: *Platform Coverage*



- Print testing example
  - Printers, spoolers, network behavior
  - Computers
  - Operating systems
  - Printer drivers

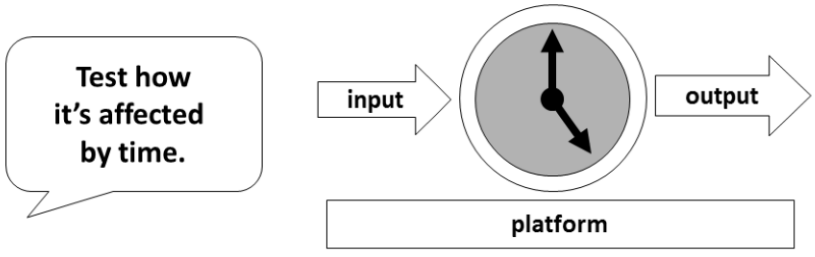
Product Elements:
Operations Coverage



- Print testing example
  - Use defaults
  - Use realistic environments
  - Use realistic scenarios
  - Use complex flows

Profiler

Product Elements:
Time Coverage



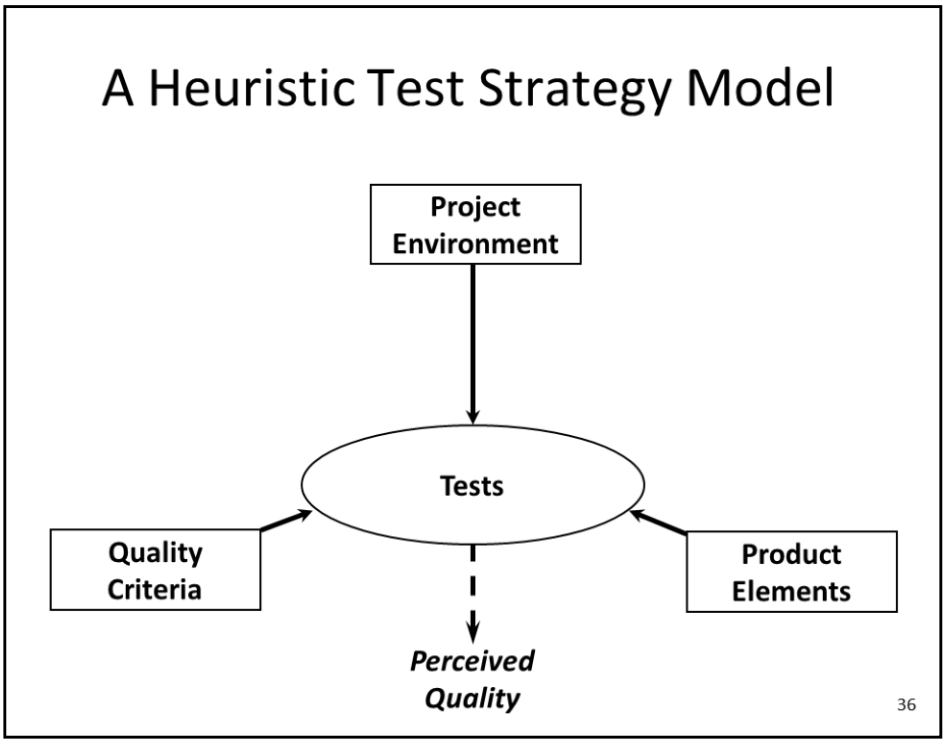
- Print testing example
  - Try different network or port speeds
  - Print one document right after another, or after long intervals
  - Try time-related constraints--spooling, buffering, or timeouts
  - Try printing hourly, daily, month-end, and year-end reports
  - Try printing from two workstations at the same time
  - Try printing again, later.

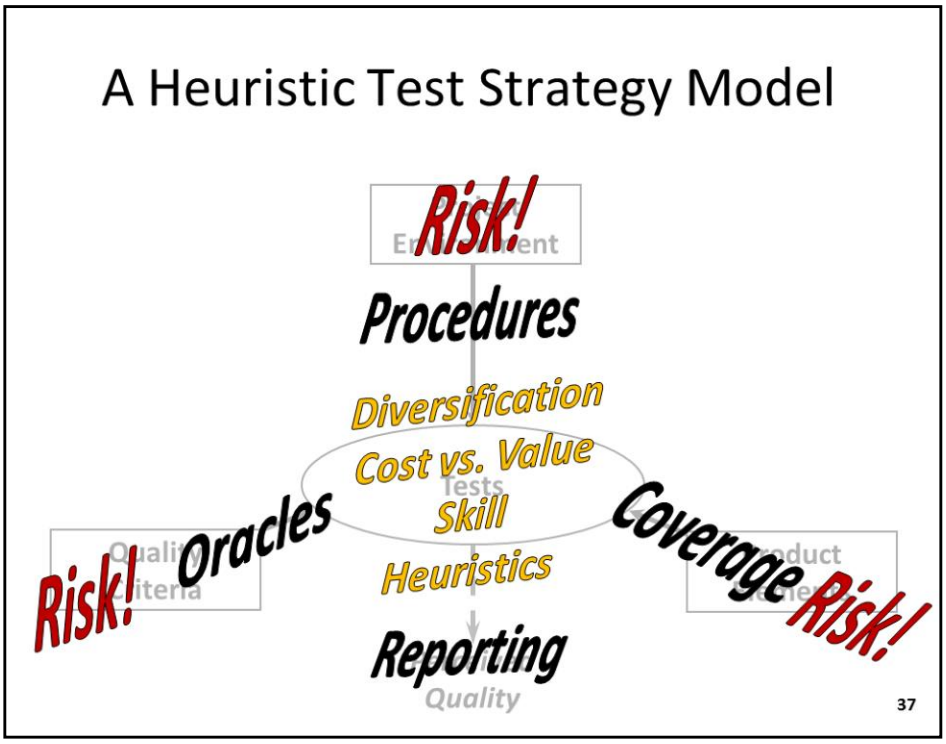
Intrinsic Testability:

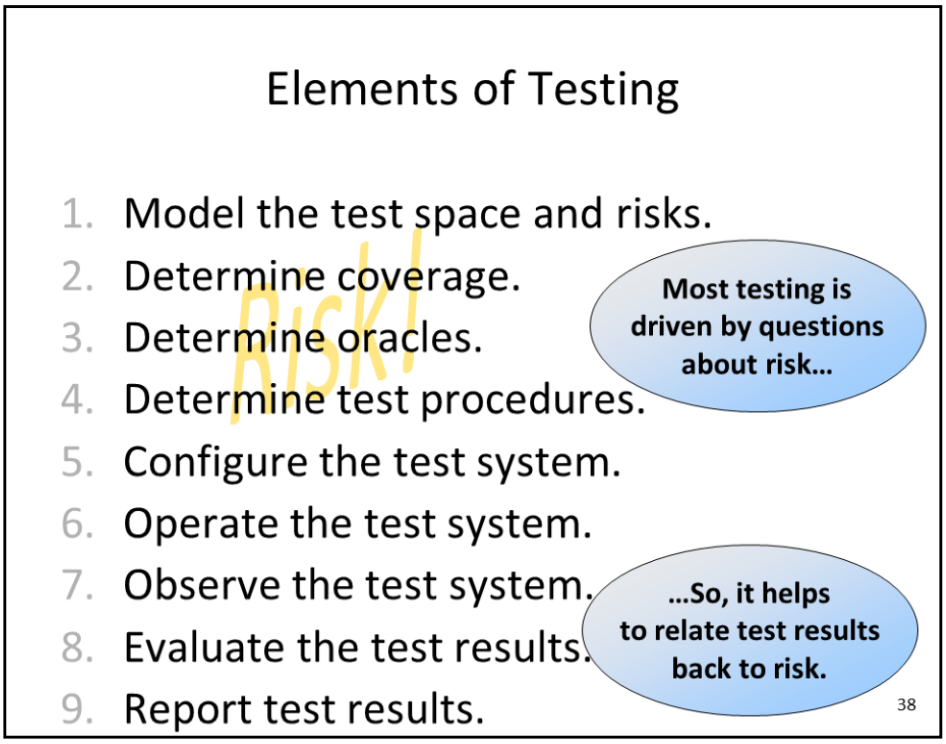
Observability & Controllability

In order to test a product well, I must be able to control the execution to visit each important state that it has, see everything important, and control the variables (in the environment) that might influence it.

Imagine a clock under a glass shield. You are not allowed to come near it or poke or probe it...

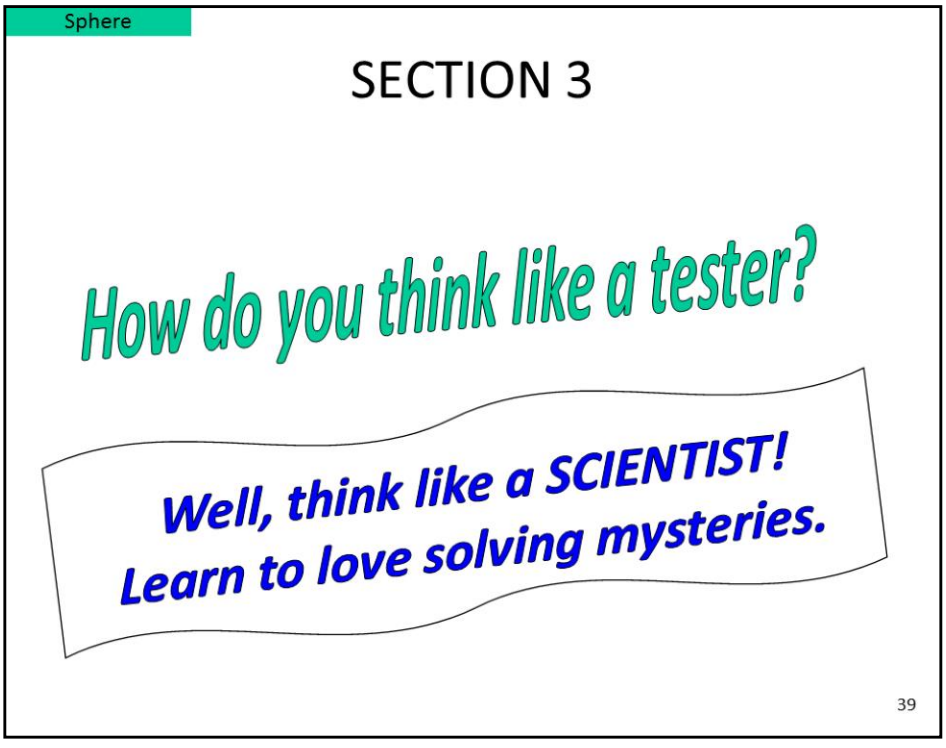




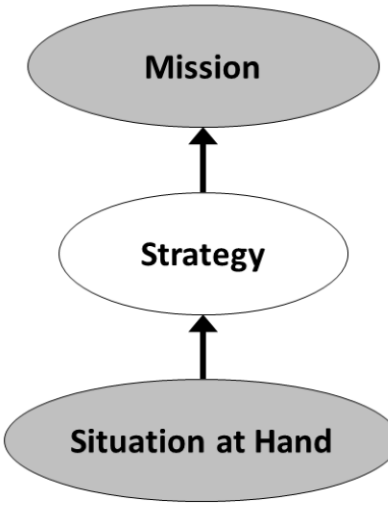


By "model", we mean that we must build a representation in our minds of what the product must do. If our model is incorrect or limited, then we automatically will not test what should be tested.

The first four items on the list are usually called "test design". The latter five items are usually called "test execution".



Don't Freeze in the Headlights!
Spot traps and avoid them.



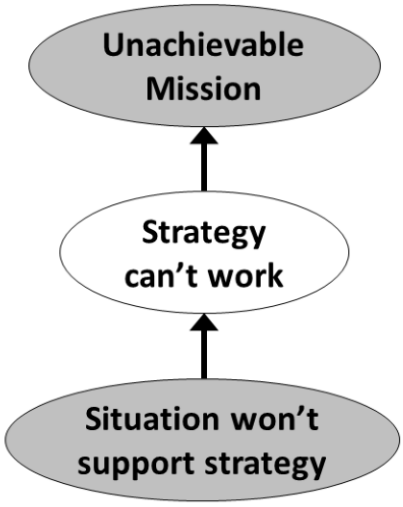
The essential purpose served by the strategy.

What we do in this situation to accomplish the mission.

The conditions under which we must work.

40

Don't Freeze in the Headlights!
Spot traps and avoid them.



Strategies

- Appear to *care*…
- Declare your assumptions.
- Ask for what you need.
- Maybe there's a simple way out.
- Share the problem; ask for help.
- Re-negotiate your mission (make a counter-offer).
- Avoid "pathetic compliance."
- Promise good faith effort, not unconditional success.

How Do We Know What "Is"?

"We know what is because we see what is."

Let's be more specific...

We believe
we know what is because we see
what we interpret as signs that indicate
what is
based on our prior beliefs about the world.

Tacit and Explicit Knowledge

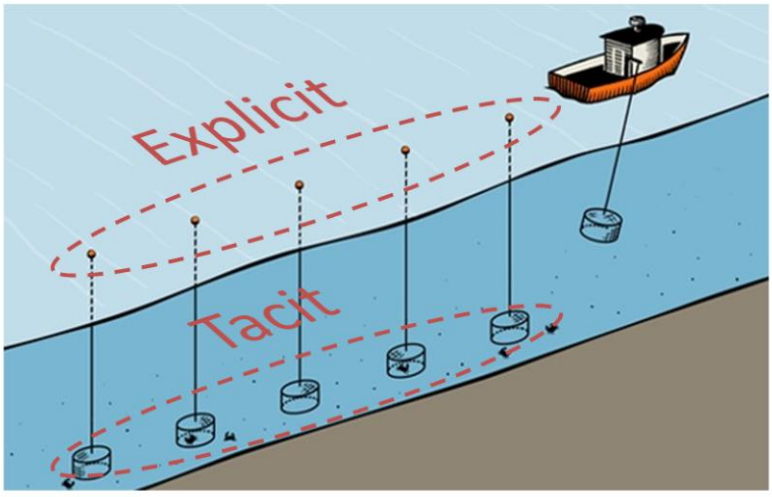
EXPLICIT means it can be represented completely in the form of a string of bits: words, pictures, even actions can be explicit. (software is explicit)

TACIT means it is not manifested in a form that can be equated to a string of bits: it is unspoken, unwritten, unpictured.

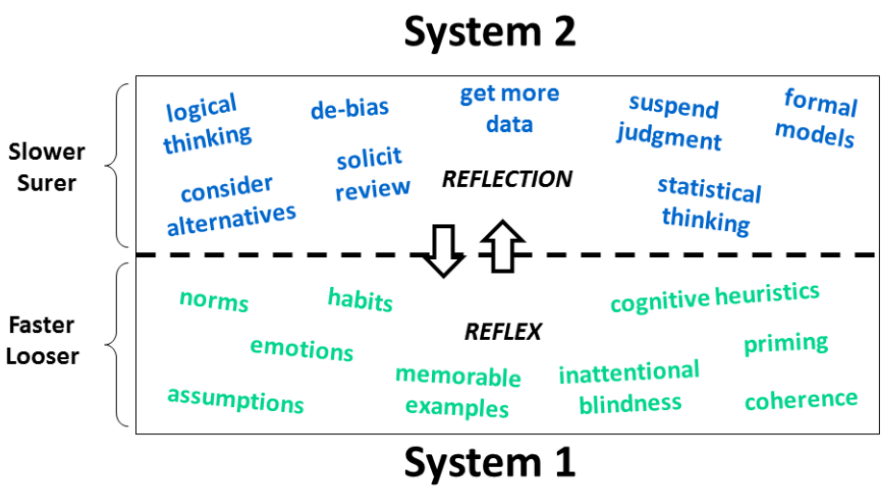
- **Relational Tacit Knowledge** is tacit by convenience.
- **Somatic Tacit Knowledge** is tacit in your body.
- **Collective Tacit Knowledge** is tacit in your community.

(see Collins, *Tacit and Explicit Knowledge*)

Managing Tacit Knowledge



Reflex is IMPORTANT
But Critical Thinking is About Reflection



See Thinking Fast and Slow, by Daniel Kahneman

Scientific Skills

- posing useful questions
- observing what's going on
- describing what you perceive
- thinking critically about what you know
- recognizing and managing bias
- designing hypotheses and experiments
- thinking despite already knowing
- analyzing someone else's thinking
- reasoning about cause and effect
- treating "facts" as merely what we *believe* we know *as of this moment*

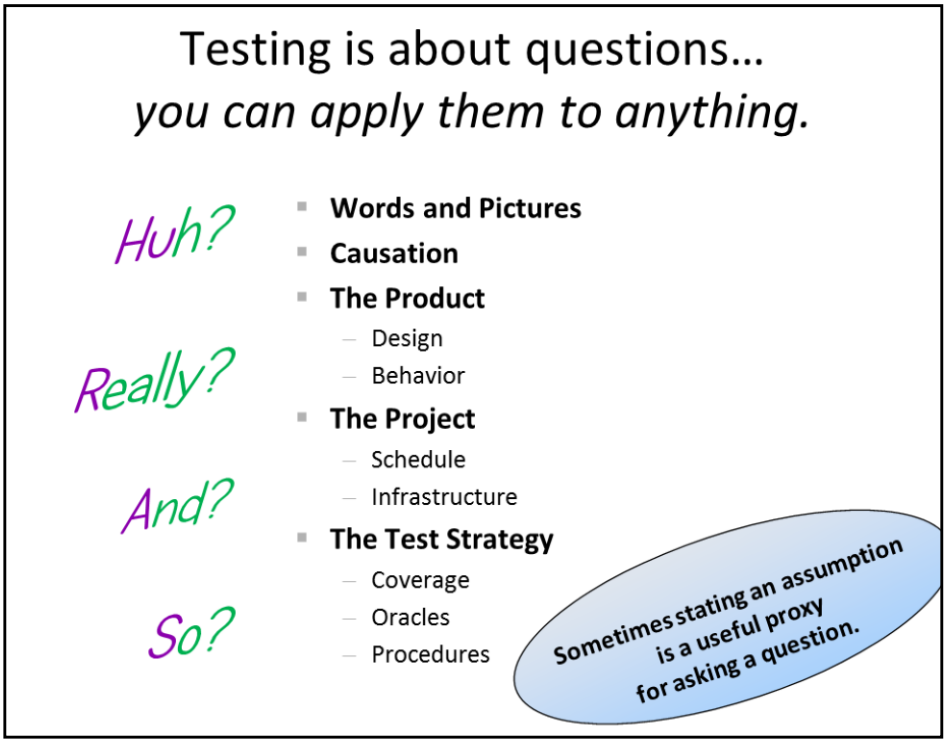
Testing is about questions...
posing them and answering them

Huh?
- You may not understand. (errors in interpreting and modeling a situation, communication errors)

Really?
- What you understand may not be true. (missing information, observations not made, tests not run)

And?
- You may not know the whole story. (perhaps what you see is not all there is)

So?
- The truth may not matter, or may matter much more than you think. (poor understanding of risk)



Thinking Like A Tester: *Seeing the Words that Aren't There*

- Among other things, *testers question premises.*
- A *suppressed premise* is an unstated premise that an argument needs in order to be logical.
- A suppressed premise is something that should be there, but isn't…
- (…or *is* there, but it's *invisible* or *implicit*.)
- Among other things, *testers bring suppressed premises to light and then question them.*
- A diverse set of models can help us to see the things that "aren't there."

Thinking Like A Tester:
Spot the missing words!

- "I performed the tests. All my tests passed. Therefore, the product works."
- "The programmer said he fixed the bug. I can't reproduce it anymore. Therefore it must be fixed."
- "Microsoft Word frequently crashes while I am using it. Therefore it's a bad product."
- "Step 1. Reboot the test system."
- "Step 2. Start the application."

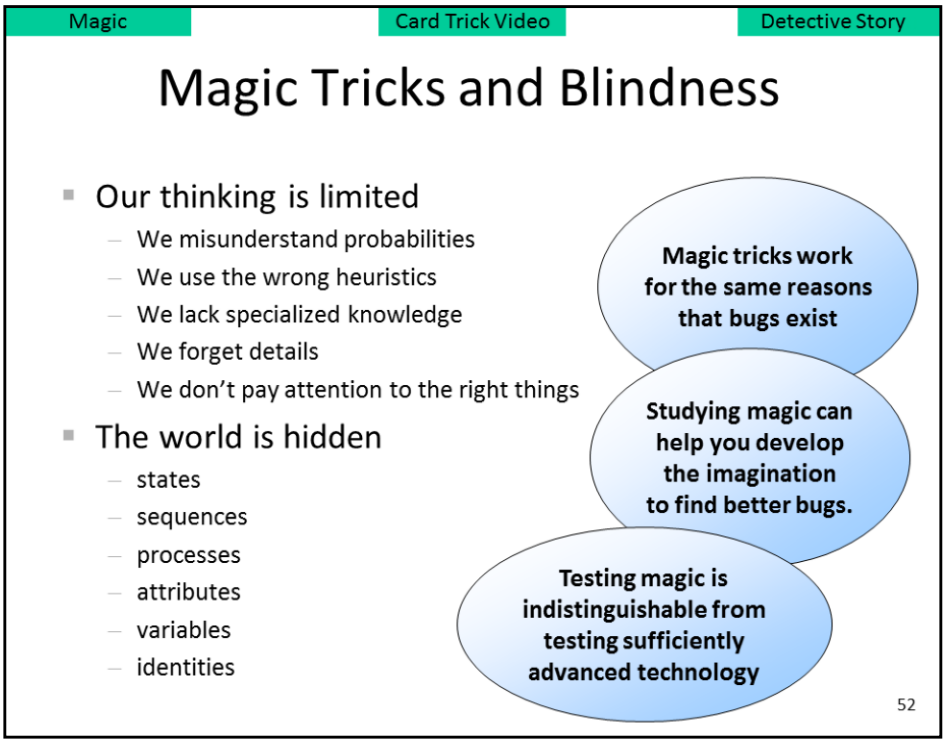
Think Critically About Requirements

How do you test this?

"The system shall operate at an input voltage range of nominal 100 - 250 VAC."

Poor answer:

"Try it with an input voltage in the range of 100-250."

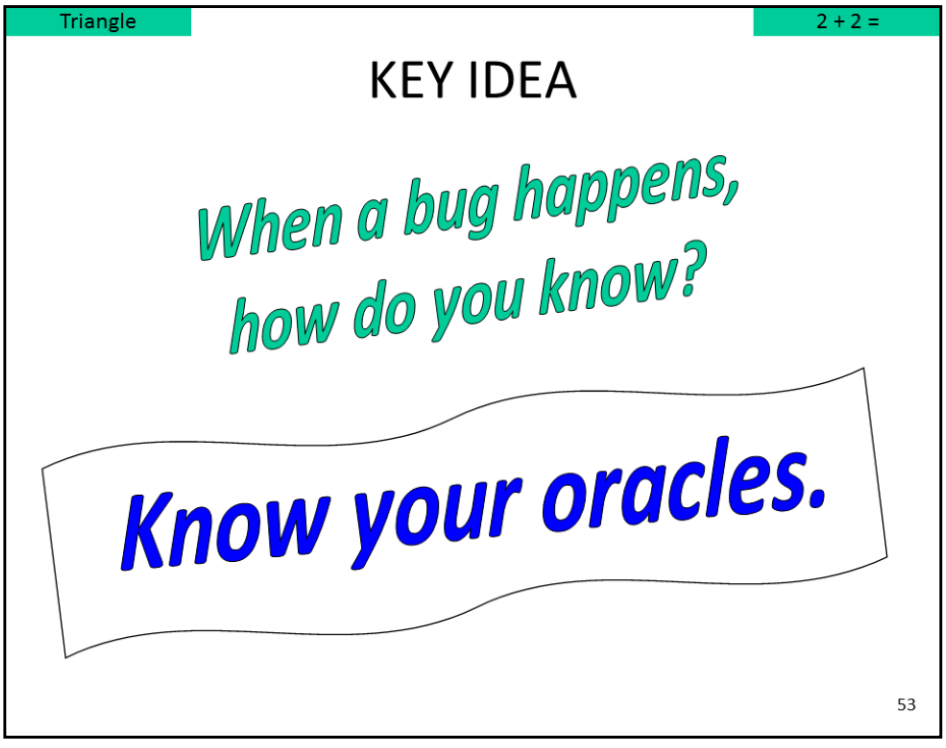


Our colleague, Jeremy Kominar, is an expert amateur magician. We asked him to provide us with some remarks on the links between software testing and magic. Here's an edited version of what he had to say--we can't quote him exactly, because he asked us not to give away his secrets!

•Learning to perform magic causes the magician to be aware of not only his own perspective, but also (and more importantly) others' perception of the trick. This becomes invaluable for testers because it helps us to recognize that the interests and perspectives of many stakeholders must be considered when testing software. The trick may "look good" or "work" from your angle, but there are always other angles to cover.

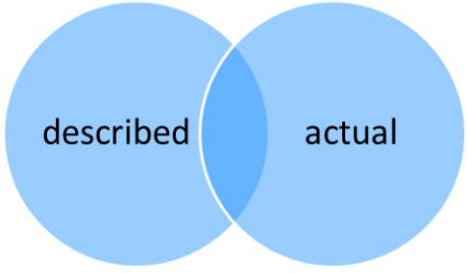
•When learning to perform magic tricks, you generally want to simplify things--minimize the number of moves and figure out how you can reach the same end result without as many steps to get there. In a testing context this idea can be used for creating different usage scenarios—taking multiple paths and variables to attain the same goal. (We might want to minimize the number of steps, but we might also want to vary the paths to shake out more bugs.)

•Learning to perform magic and learning to figure out magic tricks require us to develop observational skills, and to recognize things that deceive us. Most non-magicians fail to consider possibilities that come easily to magicians; devices, coins, or cards can be gimmicked. Non-magicians only think about normal cards. You need to think outside of the box to be a magician or to figure out how the magician does his work. This kind of reasoning and deduction are key assets to being a tester. There isn't *really* such a thing as magic but there clearly is such a thing as deception. As the magician becomes more experienced and more wise to the practice, he should be able to reverse-engineer tricks by using patterns that he already knows about magic. Once you've seen one kind of trick, similar tricks aren't so mysterious because you have heuristics that you can use to recognize how it's done.



This is what people think you do

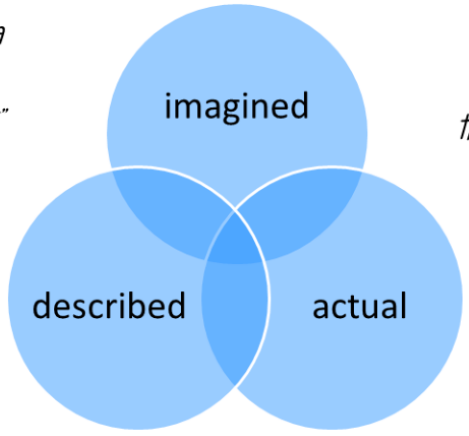
"Compare the product to its specification"



54

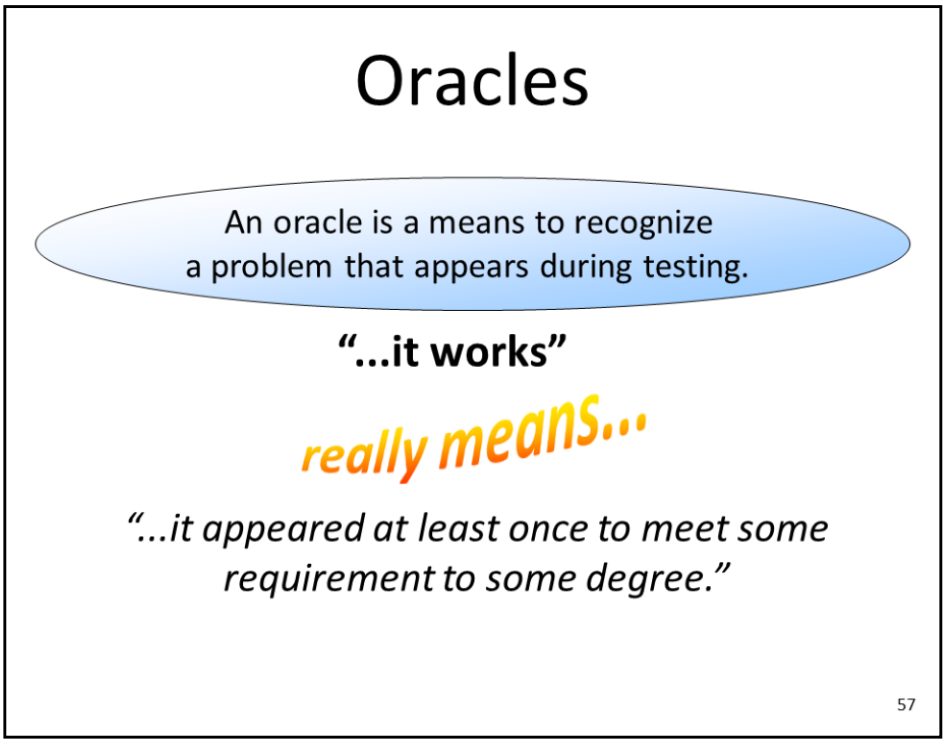
This is more like what you really do

"Compare the idea of the product to a description of it"

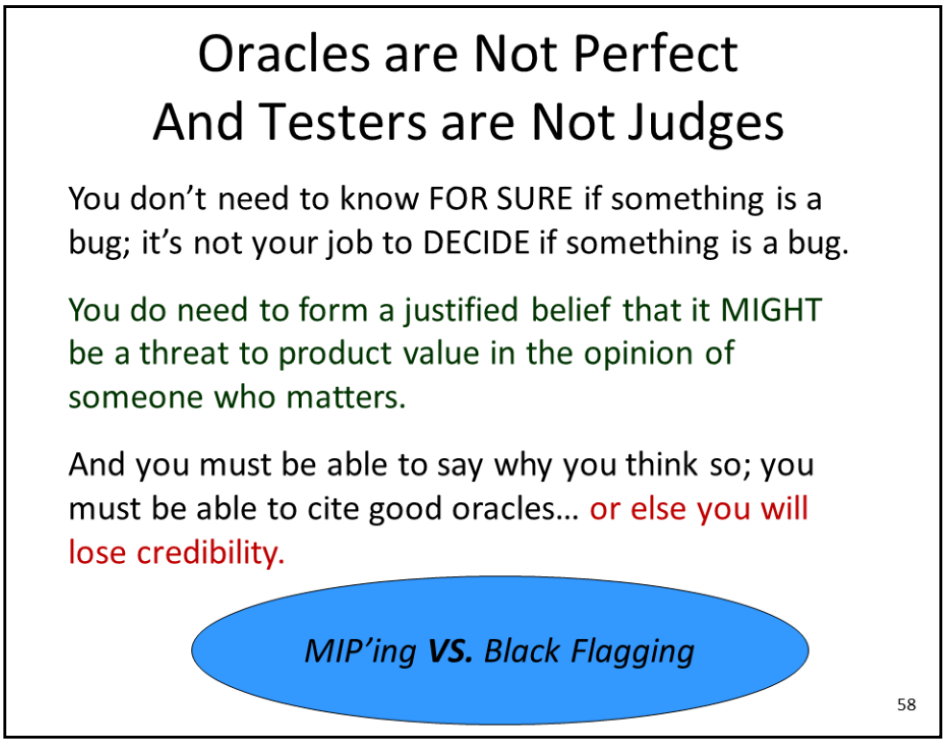


"Compare the idea of the product to the actual product"

"Compare the actual product to a description of it"



Jerry Weinberg has suggested that "it works" may mean "We haven't tried very hard to make it fail, and we haven't been running it very long or under very diverse conditions, but so far we haven't seen any failures, though we haven't been looking too closely, either." In this pessimistic view, you have to be on guard for people who say it works without checking *even once* to see if it could work.



Mipping is a way of bug reporting when you aren't confident in the value of a bug. It's an acronym that stands for "mention in passing." To MIP a bug is to report it in an informal and inexpensive way, such as through email or chat. That way, you don't get in trouble for not reporting it, and you don't get in trouble for over-reporting it.

Black flagging is the opposite of that. When you black flag a bug you put a lot of energy into the report. You may write a white paper about it, and call a meeting to discuss it. This is necessary in cases where a simple bug report won't capture the extent of the risk, or when you feel that fixing the bug will not adequately deal with the risks related to that bug (perhaps because you feel that the product may need a comprehensive re-design. Usually we black flag a bug when it seems to represent a cluster of potentially related problems that collectively pose a large risk.

General Examples of Oracles

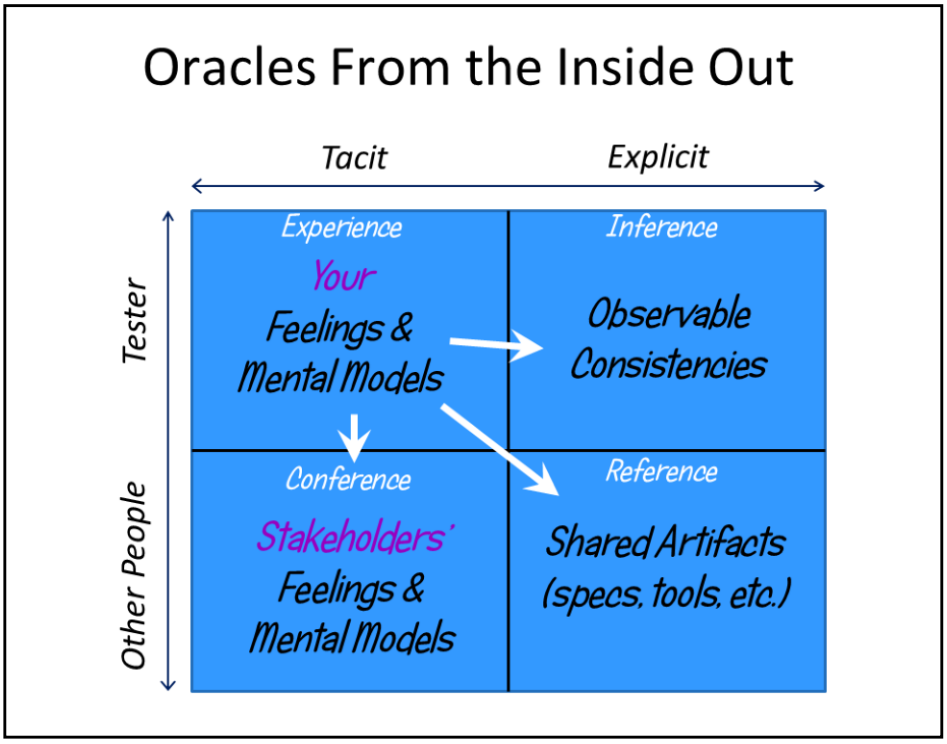
Some are more authoritative than others

- A person whose opinion matters.
- An opinion held by a person who matters.
- A disagreement among people who matter.
- A known good example output.
- A known bad example output.
- A reference document with useful information.
- A process or tool by which the output is checked.
- A process or tool that helps a tester identify patterns.
- A feeling you have, such as confusion or annoyance.
- *A desirable consistency between related things.*

59

Consistency **("this agrees with that")**
an important theme in oracles

- **Familiarity:** The product ***is not consistent*** with the pattern of any familiar problem.
- **Explainability:** The product *is consistent* with our ability to explain it.
- **World:** The product is consistent with objects and states, in the world, that it represents.
- **History:** The product *is consistent* with itself over time.
- **Image:** The product *is consistent* with an image that the organization wants to project.
- **Comparable Products:** The product *is consistent* with other products or parts of products that work in a comparable way.
- **Claims:** The product *is consistent* with what important people say it's supposed to be.
- **Users' Desires:** The product *is consistent* with what users want.
- **Product:** Each element of the product is *consistent* with comparable elements of that product.
- **Purpose:** The product *is consistent* with its purposes, both explicit and implicit.
- **Statutes & Standards:** The product *is consistent* with applicable laws and standards.



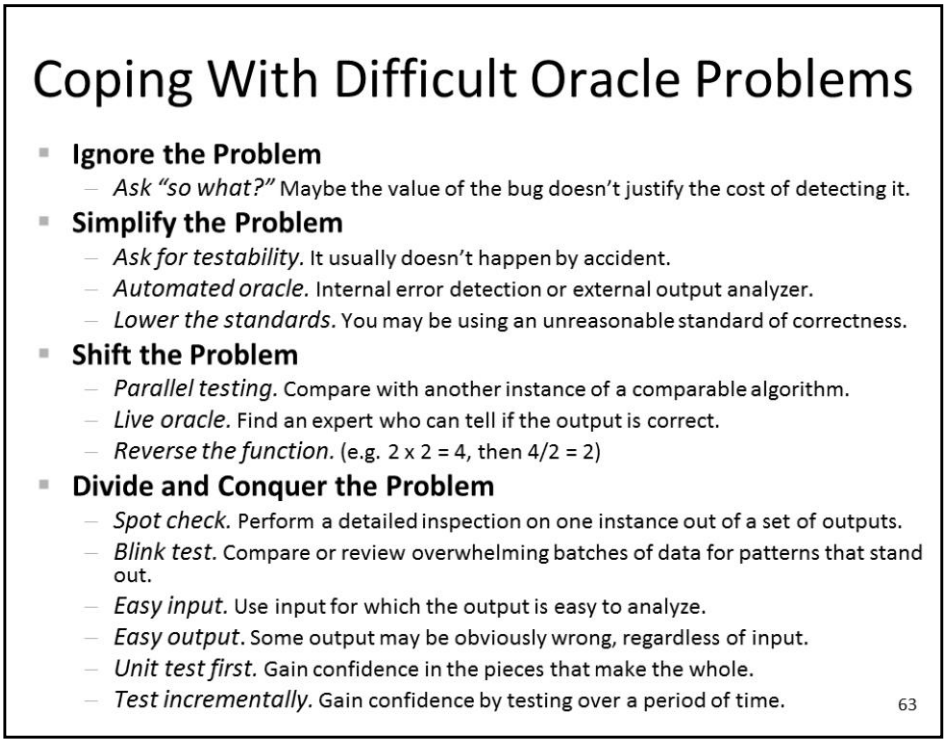
In exploratory testing, the first oracle to trigger is usually a personal feeling. Then you move from that to something more explicit and defensible. We need to move in that direction because testing is social. We probably can't just go with our gut and leave it at that, because we have to convince other people that it's really a bug.

Word Save File | Notepad Save File | Diskmapper

All Oracles Are Heuristic

- We often do not have oracles that establish a definite correct or incorrect result, in advance.
  That's why we use oracles heuristically.
- No single oracle can tell us whether a program (or a feature) is working correctly at all times and in all circumstances.
  That's why we use a variety of oracles.
- Any program that looks like it's working, to you, may in fact be failing in some way that happens to fool all of your oracles.
  That's why we proceed with humility and critical thinking.
- You (the tester) can't know the deep truth about any result.
  That's why we report whatever seems *likely* to be a bug.

62



Types of "Easy Input" oracles:

Fixed Markers. Use distinctive fixed input patterns that are easy to spot in the output.

Statistical Markers. Use populations of data that have distinguishable statistical properties.

Self-Referential Data. Use data that embeds metadata about itself. (e.g. counterstrings)

Easy Input Regions. For specific inputs, the correct output may be easy to calculate.

Outrageous Values. For some inputs, we expect error handling.

Idempotent Input. Try a case where the output will be the same as the input.

Match. Do the "same thing" twice and look for a match.

Progressive Mismatch. Do progressively differing things over time and account for each difference. (code-breaking technique)

Fixed Markers. Use distinctive fixed input patterns that are easy to spot in the output. Some programmers use the hex values 0xDEADBEEF as a flag to indicate uninitialized data.

Statistical Markers. Use populations of data that have distinguishable statistical properties. If you create input that has a specific statistical property, then you may be able to detect missing or corrupted data by examining the statistical properties of the output.

Self-referential data. Use data that embeds data about itself. (e.g. counterstrings). *Counterstrings* are strings that identify their lengths. *3*5*7*9*12*15* is an example of a counterstring. (Satisfice's PERLCLIP tool creates counterstrings of arbitrary lengths.) Another example is using JAMES_FIRSTNAME and BACH_LASTNAME as sample data in the first name and last name fields of a database.

Easy Input Regions. For specific inputs, the correct output may be easy to calculate. A number times one is always itself; a number plus zero is always itself; a multiple of ten always ends in 0, and so on.

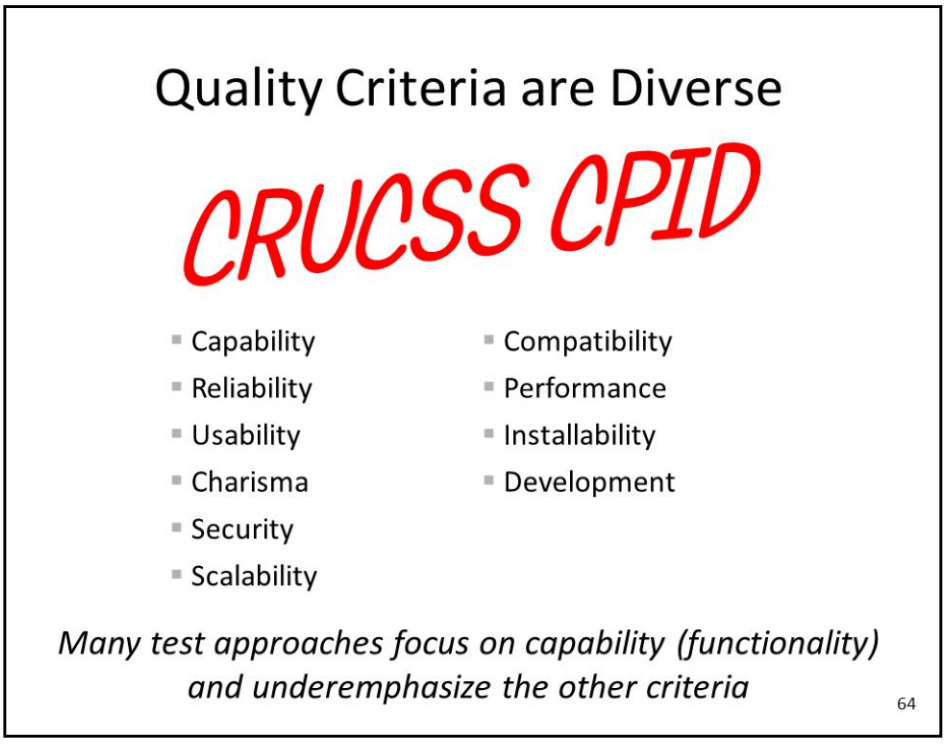
Outrageous values. For some inputs, we expect error handling. Examples include letters in fields that expect

only numbers, extremely long strings in fields that expect short ones, huge numbers where modest ones are expected, or zero or null values where some positive value is required. Each example should trigger some kind of error handling.

Idempotent input. Try a case where the output will be the same as the input. Take some data and process it in some way, such that the output from the first run can be used as input for subsequent runs but shouldn't go through additional changes. Examples include spell-checking (after all the corrections have been accepted for the first run, no further changes should be suggested) or code formatters (after the code has been formatted once, the same code should pass through the process unchanged).

Match. Do the "same thing" twice and look for a match. Given the same input data and the same function on that data, we would typically expect the output to be the same. This approach is sometimes used to check various kinds of encoding or checksums.

Progressive mismatch. Do progressively differing things over time and account for each difference. Vary the input in some limited or controlled way. This is an exploratory approach, sometimes used as a code-breaking technique.



Development. *How well can we create, test, and modify it?*

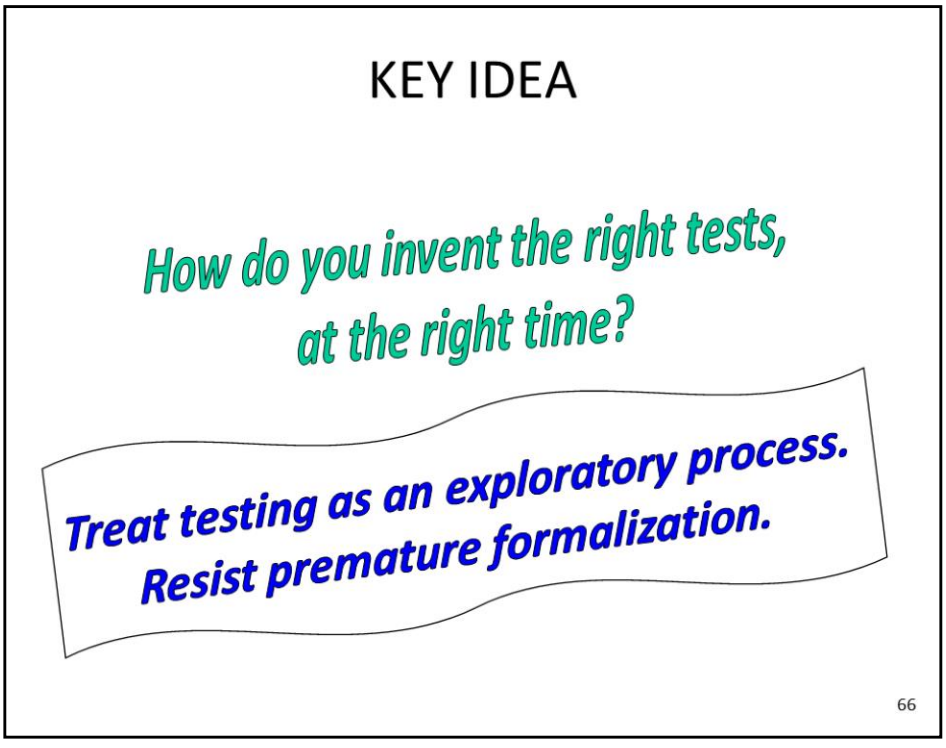
- *Supportability:* How economical will it be to provide support to users of the product?
- *Testability:* How effectively can the product be tested?
- *Maintainability:* How economical is it to build, fix or enhance the product?
- *Portability:* How economical will it be to port or reuse the technology elsewhere?
- *Localizability:* How economical will it be to adapt the product for other places?

A General Statement About Oracles to Discourage Silly Documentation

3.0 TEST PROCEDURES

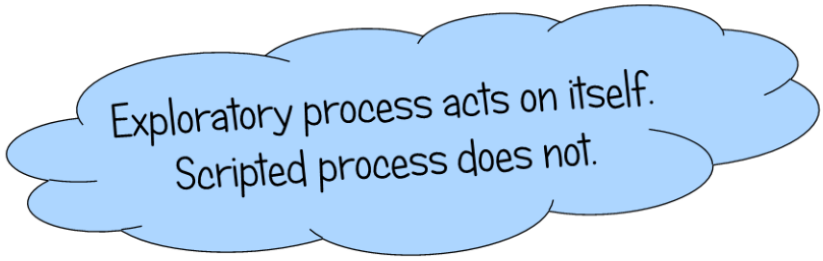
3.1 **General Testing Protocol**

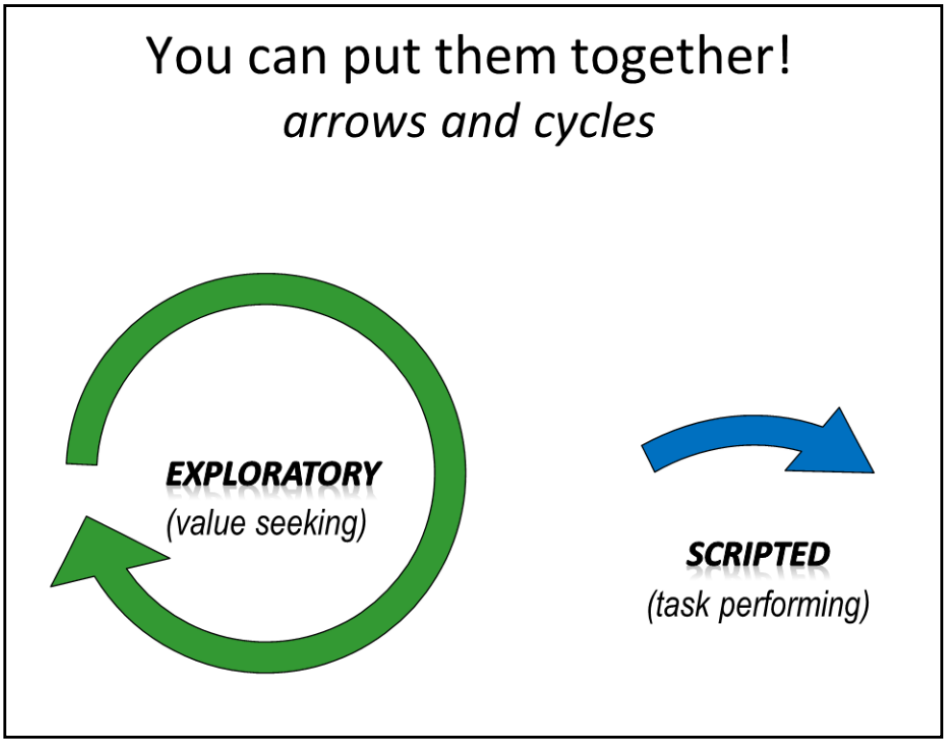
- *In the test descriptions that follow, the word "verify" is used to highlight specific items that must be checked.* In addition to those items, the tester shall be at all times alert for any unexplained or erroneous behavior of the product. The tester shall bear in mind that, regardless of any specific requirement for any specific test, there is the overarching general requirement that the product shall not pose an unacceptable risk of harm to the patient, including an unacceptable risk due to reasonably foreseeable misuse.
- *Test personnel requirements:* The tester shall be thoroughly familiar with the Generator and Workstation FRS, as well as with the working principles of the devices themselves. The tester shall also know the working principles of the power test jig and associated software, including how to configure and calibrate it and how to recognize if it is not working correctly. The tester shall have sufficient skill in data analysis and measurement theory to make sense of statistical test results. The tester shall be sufficiently familiar with test design to complement this protocol with exploratory testing in the event that anomalies appear that require investigation. The tester shall know how to keep test records to a credible professional standard.



Exploratory Process means

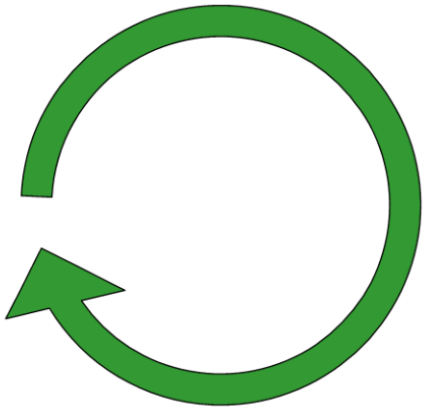
Self-Controlled Seeking in a Complex Environment.

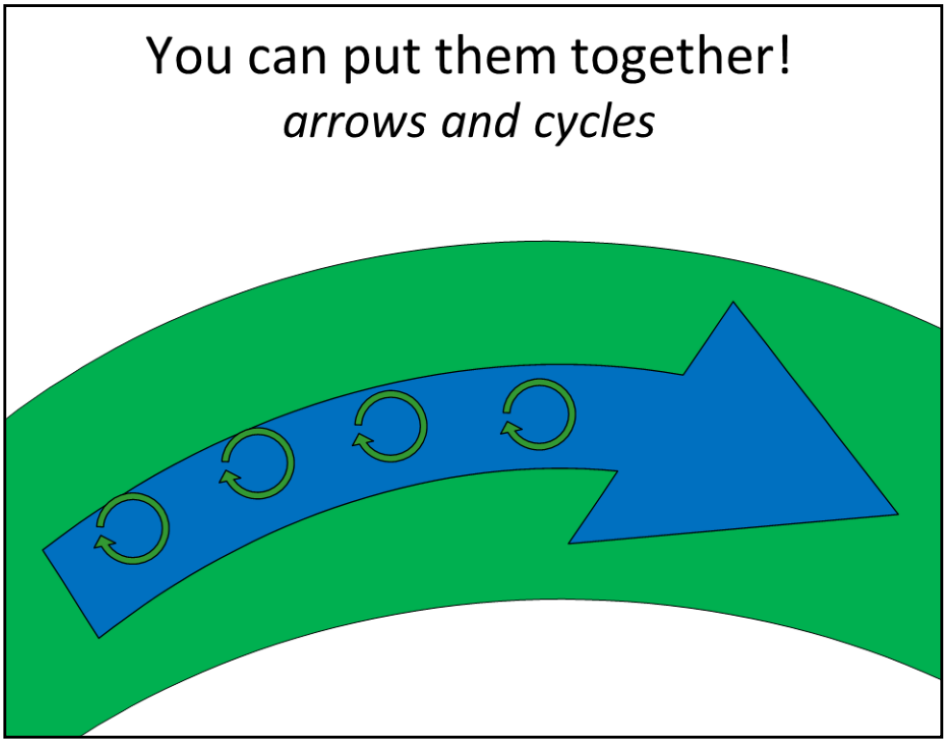




You can put them together!

arrows and cycles

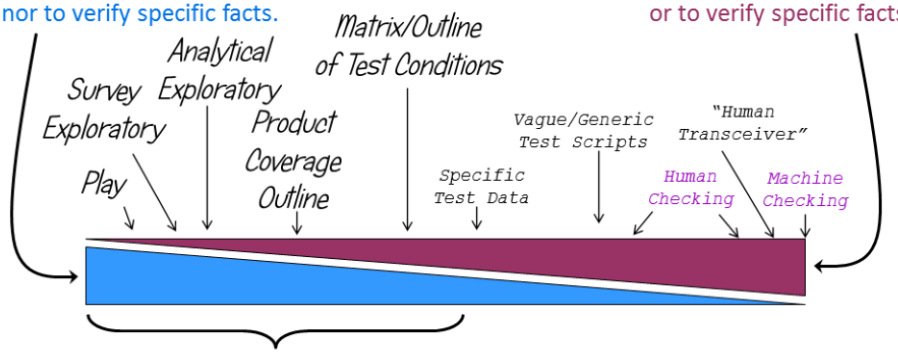




The Testing Formality Continuum
Mixing Scripting and Exploration

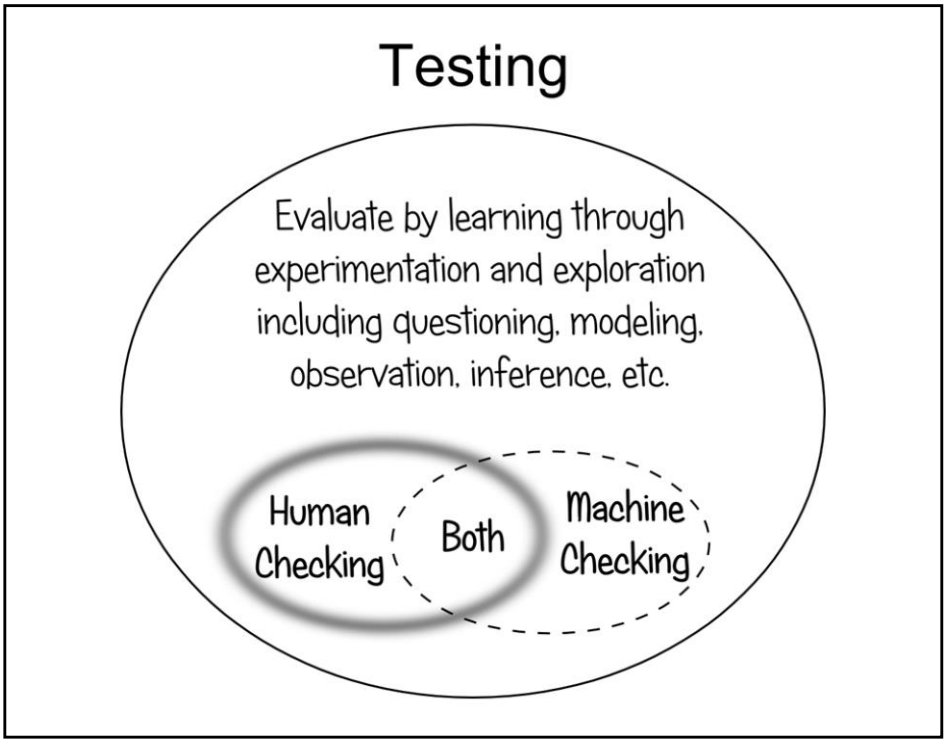
INFORMAL
Not done in any specific way, nor to verify specific facts.

FORMAL
Done in a specific way, or to verify specific facts.



When I say "exploratory testing" and don't qualify it, I mean anything on the informal side of this continuum.

72



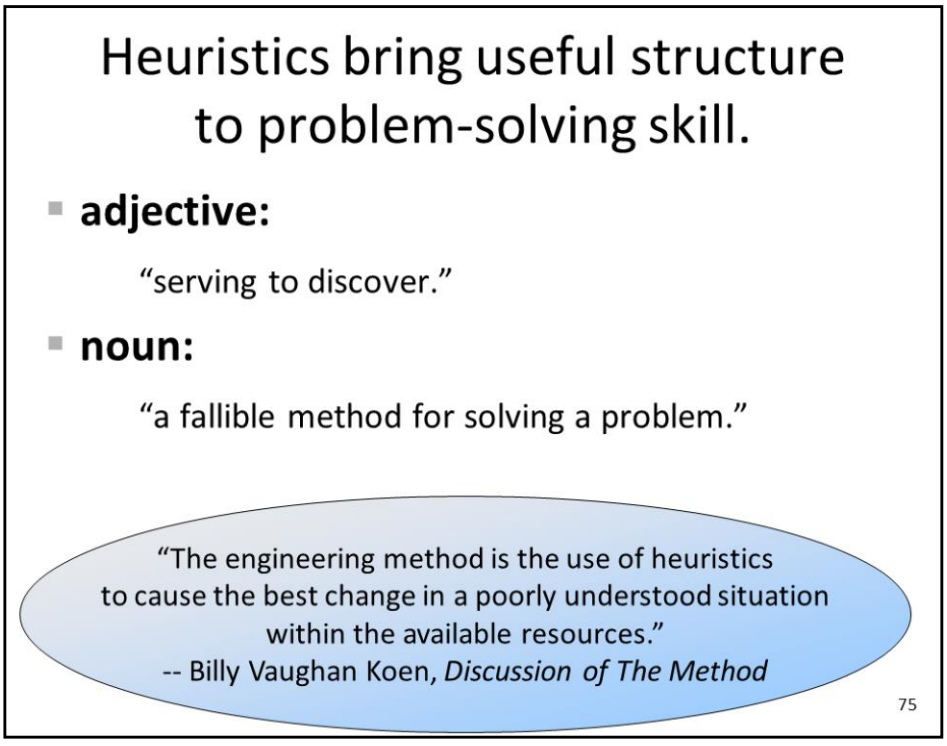
IP Address

Exploratory Process is Structured

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

In other words, it's not "random", but systematic.

74



Key ideas about heuristics:

- A heuristic is not an *edict*. Heuristics require guidance and control of skilled practitioner.
- Heuristics are context-dependent.
- Heuristics may be useful even when they contradict each other– especially when they do!
- Heuristics can substitute for complete and rigorous analysis.

Types of heuristics:

- Guideword Heuristics: Words or labels that help you access the full spectrum of your knowledge and experience as you analyze something.

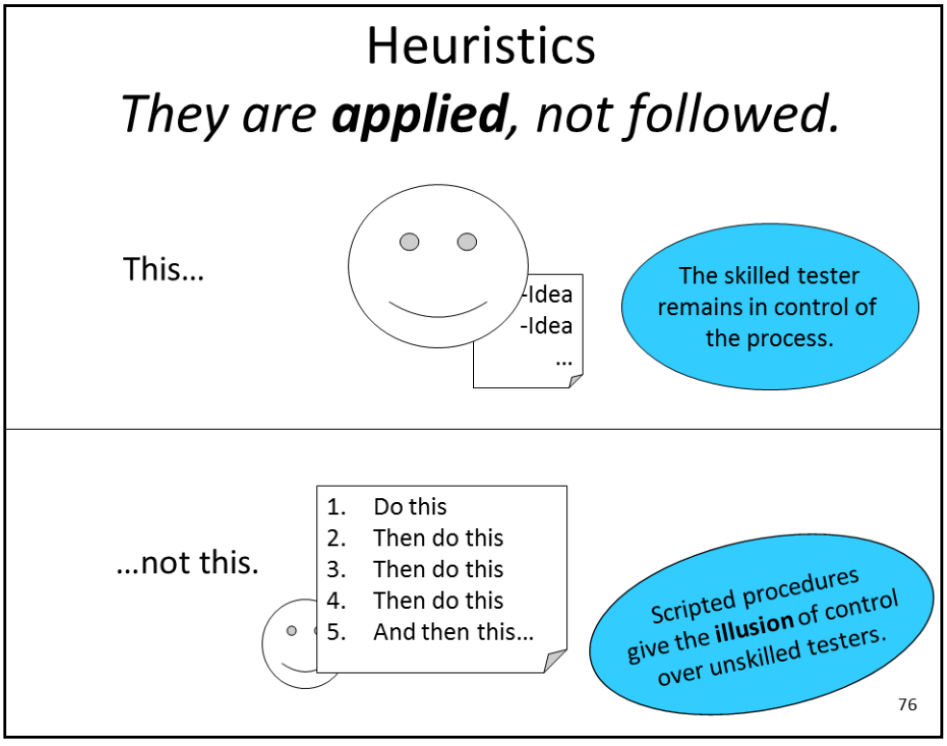
- Trigger Heuristics: Ideas associated with an event or condition that help you recognize when it may be time to take an action or think a particular way. Like an alarm clock for your mind.

- Subtitle Heuristics: Help you reframe an idea so you can see alternatives and bring out assumptions during a conversation.

- Heuristic Model: A representation of an idea, object, or system that helps you explore, understand, or control it.

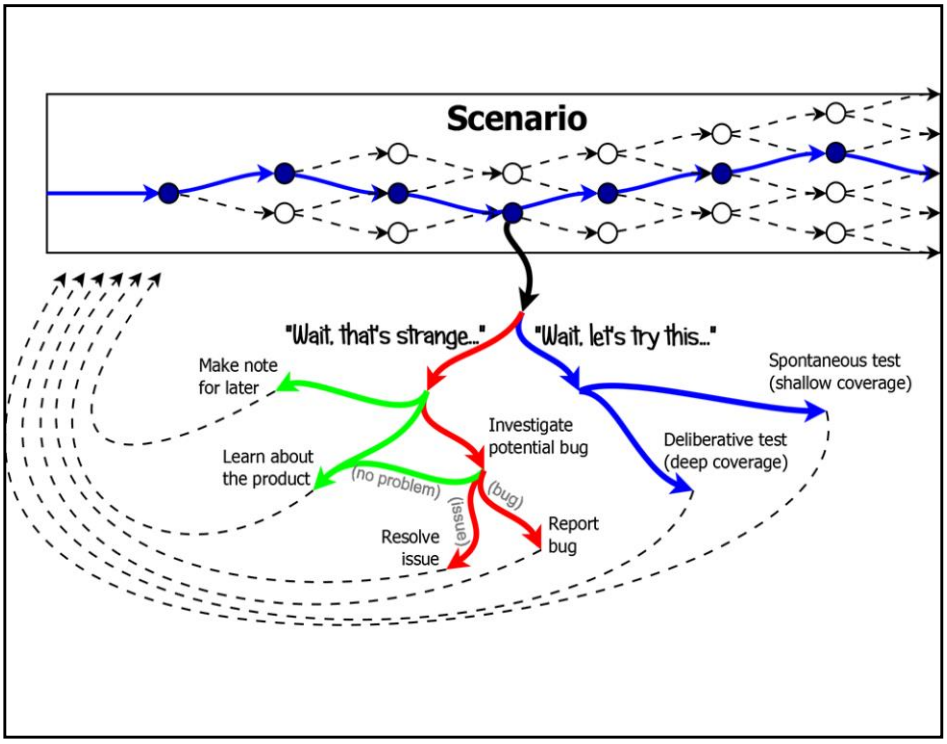
- Heuristic Procedure or Rule: A plan of action that may help solve a class of problems.

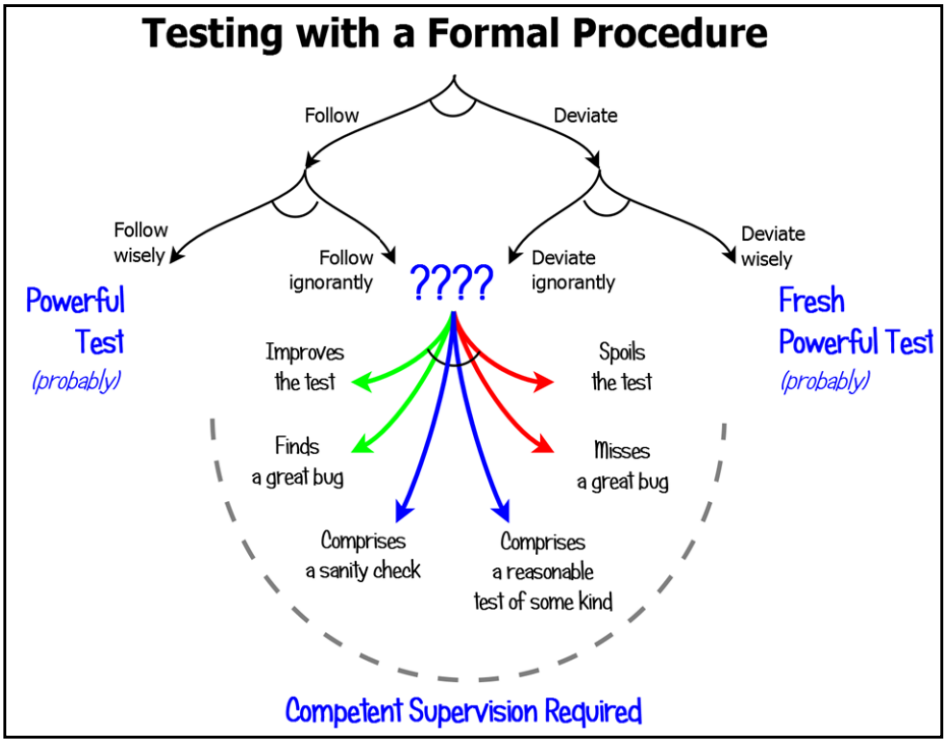
•Cognitive Heuristics: Heuristics built into your mind by virtue of how human brains work. These show themselves as systematic biases in our senses (optical illusions) and decision-making.



Everyday examples of heuristics:

- It's dangerous to drink and drive.
- A bird in hand is worth two in the bush.
- Nothing ventured, nothing gained.
- Sometimes people stash their passwords near their computers. Try looking there.
- Stores are open later during the Holidays.
- If your computer is behaving strangely, try rebooting. If it's very strange, reinstall Windows.
- If it's a genuinely important task, your boss will follow-up, otherwise, you can ignore it.



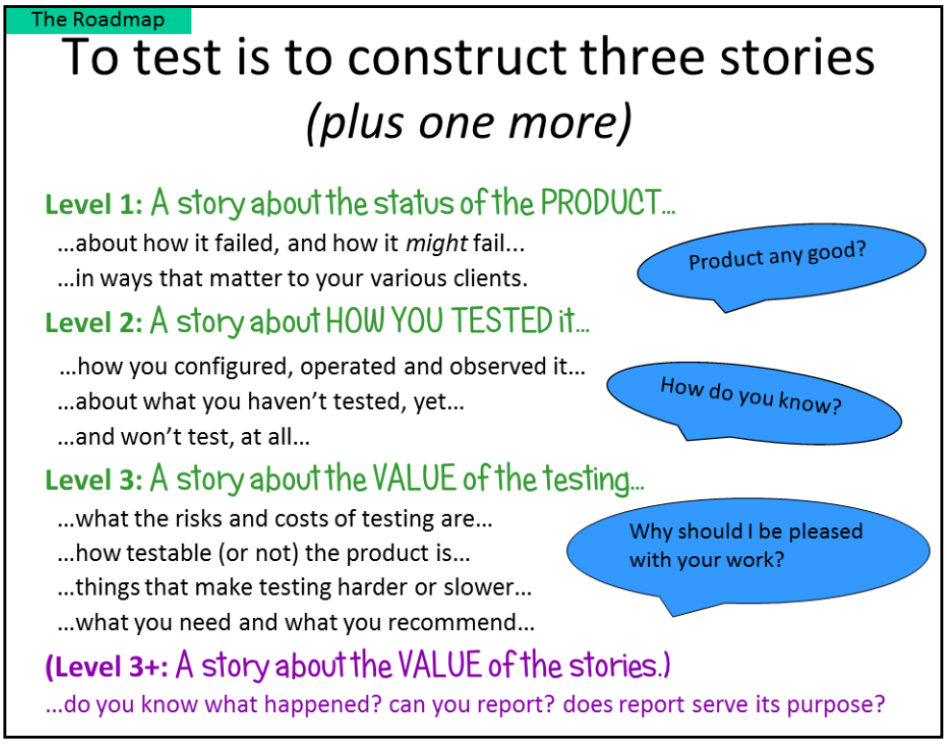


Pen Test

ET is a Structured Process

In excellent exploratory testing, one structure tends to dominate all the others:

Exploratory testers construct a compelling story of their testing. It is this story that gives ET a backbone.

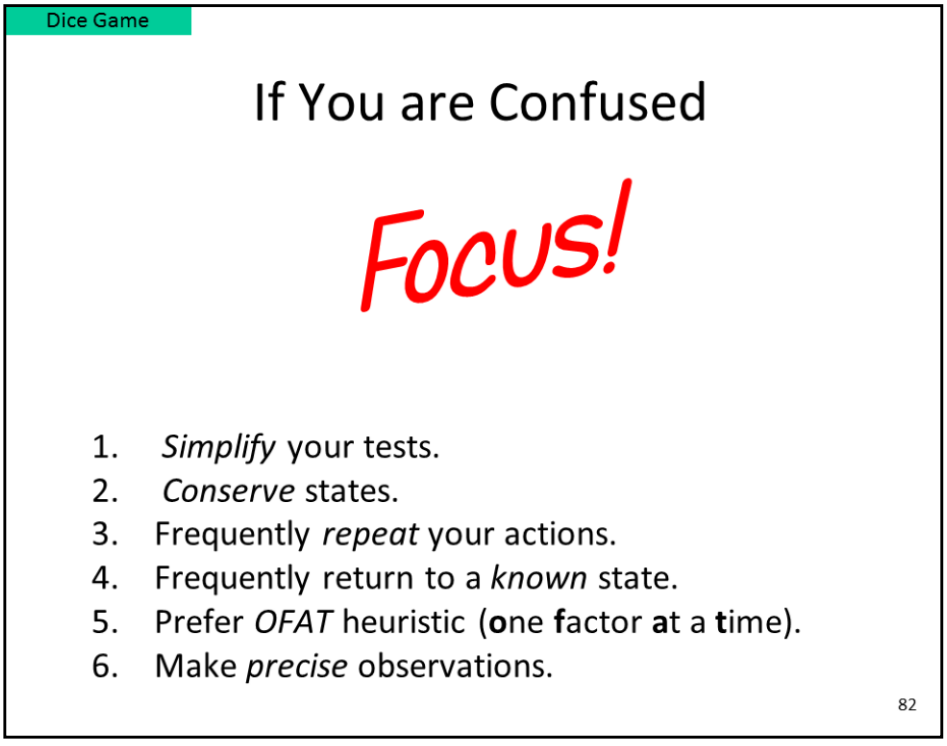


Boundary Testing

If You are Frustrated

De-Focus!

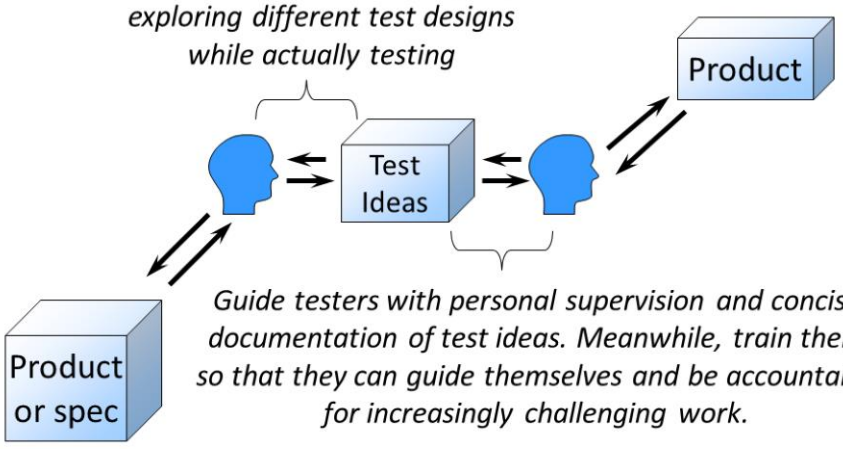
1. Look over your recent tests and find a pattern there.
2. With your next few tests, *violate* the old pattern.
3. Prefer *MFAT* (**m**ultiple **f**actors **a**t a **t**ime).
4. Broaden and vary your observations.



"State Conservation" means that you minimize disturbances to the product state, maintaining it as much as possible while using the product.

Test Design and Execution

Achieve excellent test design by exploring different test designs while actually testing



Guide testers with personal supervision and concise documentation of test ideas. Meanwhile, train them so that they can guide themselves and be accountable for increasingly challenging work.

83

Allow some *disposable time*
Self-management is good!

- How often do you account for your progress?

Every minute?
Every hour?
Every day?

- If you have *any autonomy at all*, you can risk investing *some time* in
  - learning
  - thinking
  - refining approaches
  - better tests

Allow some *disposable time*

If it turns out that you've made a bad investment…*oh well*

If it turns out that you've made a *good* investment, you might have

- learned something about the product
- invented a more powerful test
- found a bug
- done a better job
- avoided going down a dead end for too long
- surprised and impressed your manager

85

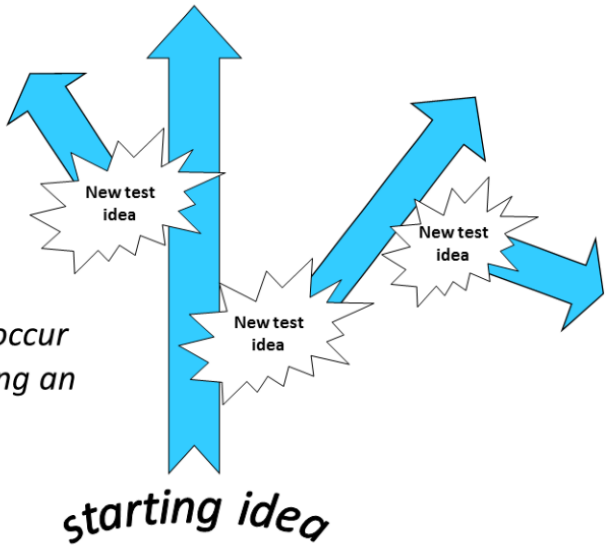
"Plunge in and Quit" Heuristic

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

- This benefits from *disposable time*– that part of your work not scrutinized in detail.
- Plunge in and quit means you can start something without *committing* to finish it successfully, and therefore *you don't need a plan*.
- Several cycles of this is a great way to *discover* a plan.

86

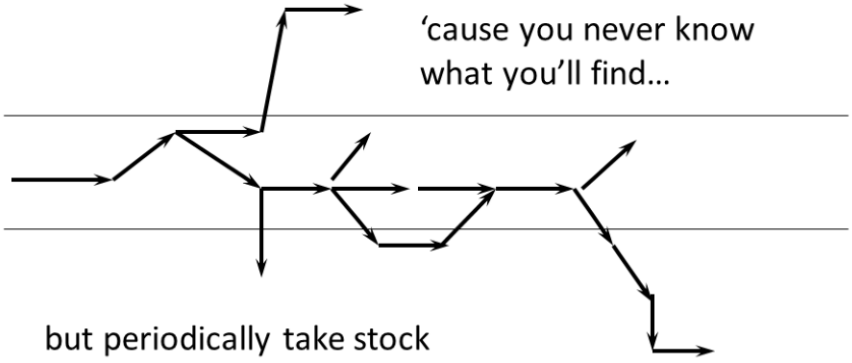
Exploratory Branching:
Distractions are good!



New test ideas occur continually during an ET session.

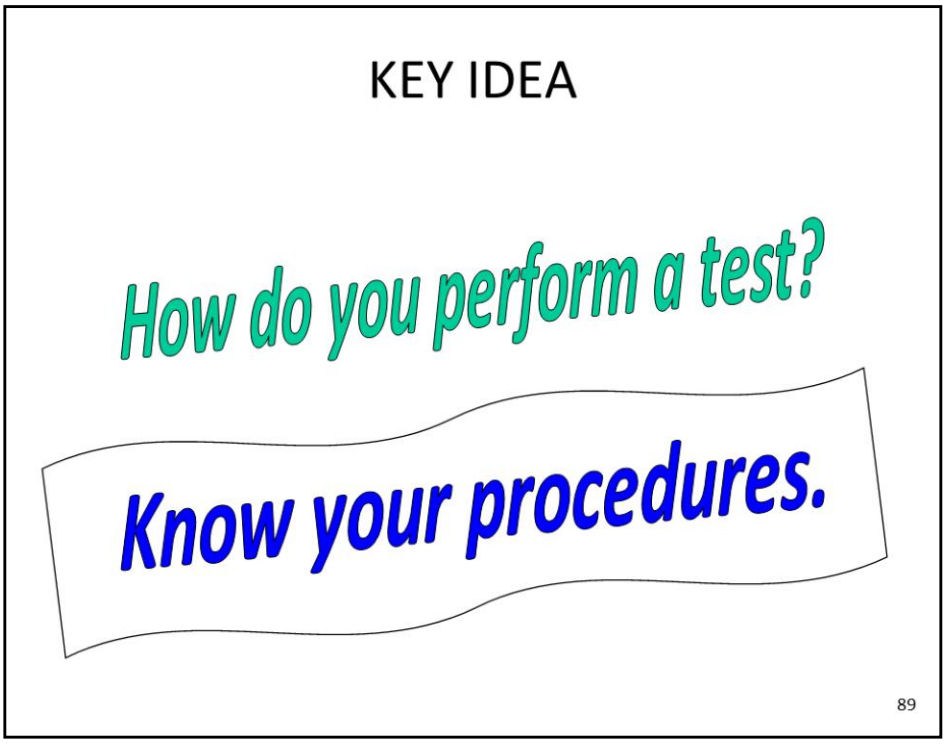
"Long Leash" Heuristic

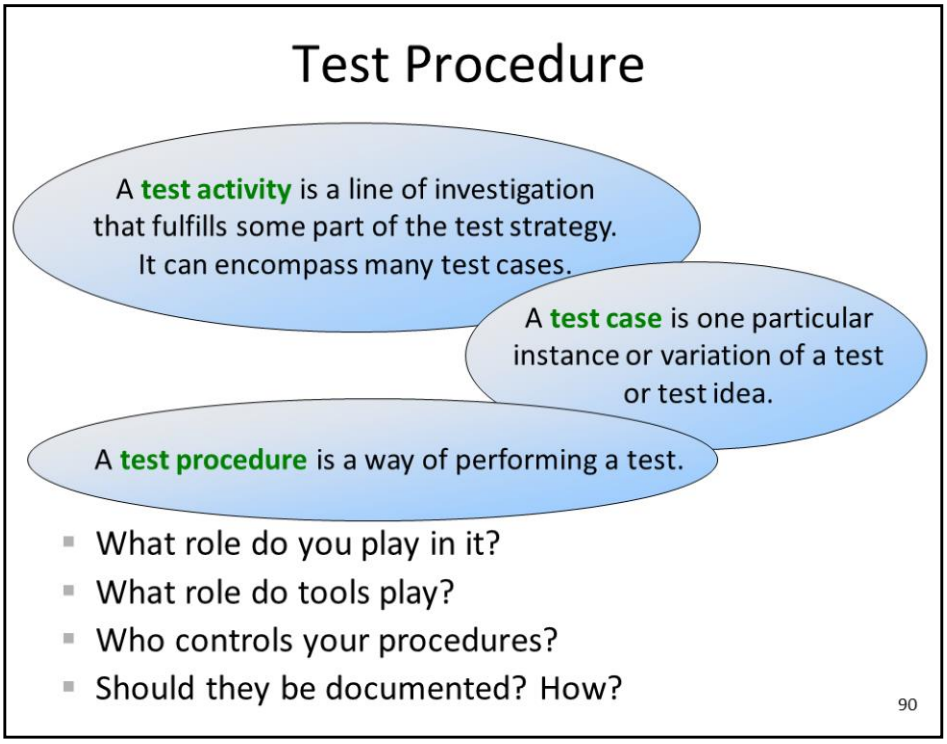
Let yourself be distracted…



but periodically take stock
of your status against your mission

88





Some people prefer a more restrictive definition of procedure. Cem Kaner considers a sequential set of steps to be an important aspect of procedure. He would prefer that we use the word "test implementation" instead of procedure, in this section.

However, in the Rapid Testing methodology, we usually opt for the most inclusive definitions that still express the key ideas, since that makes the conversation easier. In our view, the word procedure as it is commonly used is not limited only to sequential instruction sets.

As always, it's up to you what words you use.

Wordpad

Test Procedure *has four elements*

Addresses a motivating question

- **Configure**
  - *(if necessary)* Obtain product for testing
  - *(if necessary)* Install on a platform.
  - *(if necessary)* Prepare test data and tools for test execution.
  - Assure that the product is in a "clean enough" starting state.
- **Operate**
  - Control the product and platform with inputs to exercise the product.
  - Cause the product to exercise the right functions/states in the right sequence with the right data.
- **Observe**
  - Collect information about how the product behaves (collect both direct and indirect output) so that it can be evaluated.
- **Evaluate**
  - Apply oracles to detect bugs.

Provides a clear answer to the question

91

To maximize test *integrity*…

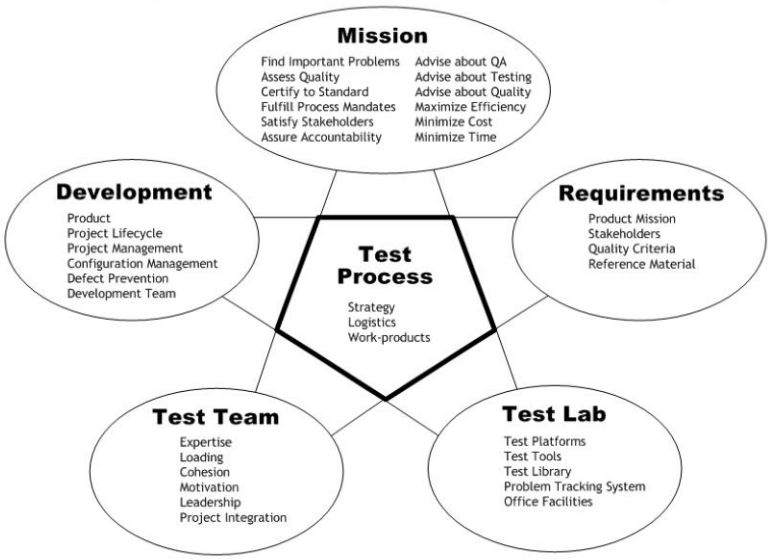
1. Start the test from a *known* (clean) state.
2. Prefer *simple, deterministic* actions.
3. Trace test steps to a *specified model*.
4. Follow *established and consistent* lab procedures.
5. Make *specific* predictions, observations and records.
6. Make it *easy to reproduce* (automation helps).

To find *unexpected problems*,
elusive problems that occur in sustained field use,
or more problems *quickly* in a complex product…

1. Start from *different states* (not necessarily clean).
2. Prefer *complex, challenging* actions.
3. Generate tests from a *variety* of models.
4. *Question* your lab procedures and tools.
5. Try to *see everything* with open expectations.
6. Make the test *tougher*, instead of more reproducible.

The Test Project Context

Where are you and what are you doing there?



95

Test Strategy

- Strategy: "The set of ideas that guide your **test design** or **choice of tests** to be performed."
- Logistics: "The set of ideas that guide your **application of resources** to fulfilling the test strategy."
- Plan: "The set of ideas that guide your **test project**."
- A good test strategy is:
  - *Product-Specific*
  - *Risk-focused*
  - *Diversified*
  - *Practical*

plan = strategy + logistics

Visual

One way to make a strategy…

1. Learn the product.
2. Think of important potential problems.
3. Think of how to search the product for *those problems*.
4. Think of how to search the product, *in general*.

Think of ways that:

- will take advantage of the *resources* you have.
- comprise a *mix* of different techniques.
- comprise something that you really *can* actually do.
- serve the *specific* mission you are expected to fulfill.

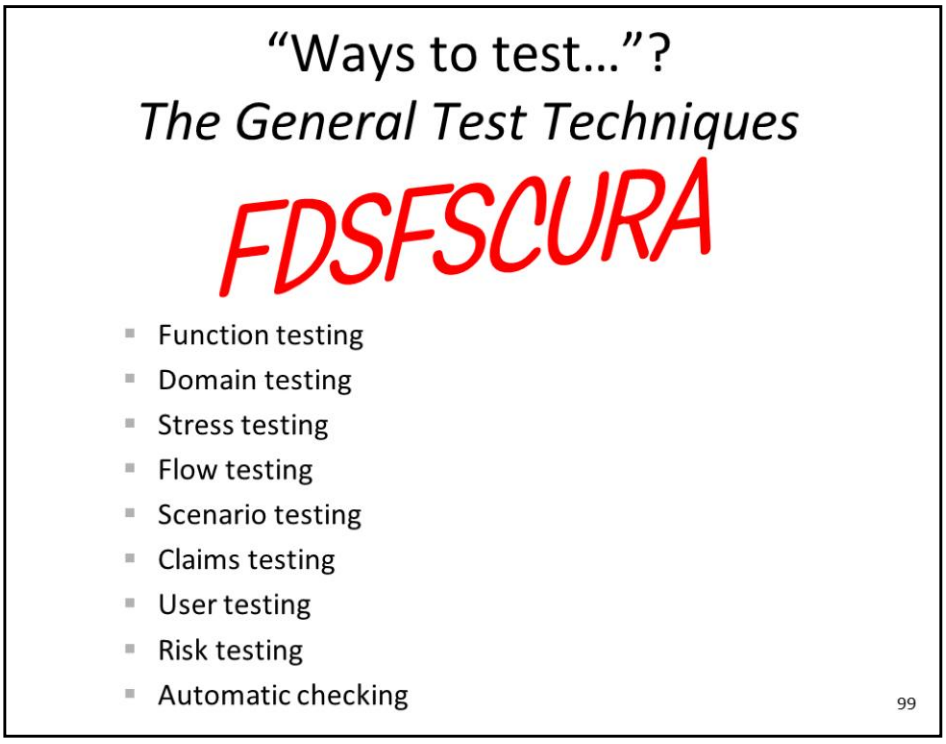
97

What does "taking advantage of resources" mean?

MIDTESTD

- Mission
  - *The problem we are here to solve for our customer.*
- Information
  - *Information about the product or project that is needed for testing.*
- Developer relations
  - *How you get along with the programmers.*
- Team
  - *Anyone who will perform or support testing.*
- Equipment & tools
  - *Hardware, software, or documents required to administer testing.*
- Schedule
  - *The sequence, duration, and synchronization of project events.*
- Test Items
  - *The product to be tested.*
- Deliverables
  - *The observable products of the test project.*

98



We'll come back to these later. For now, note that we're more likely to catch different bugs with different nets; a diverse set of test techniques will find a diverse set of bugs.

How much is enough?
Diverse Half-Measures

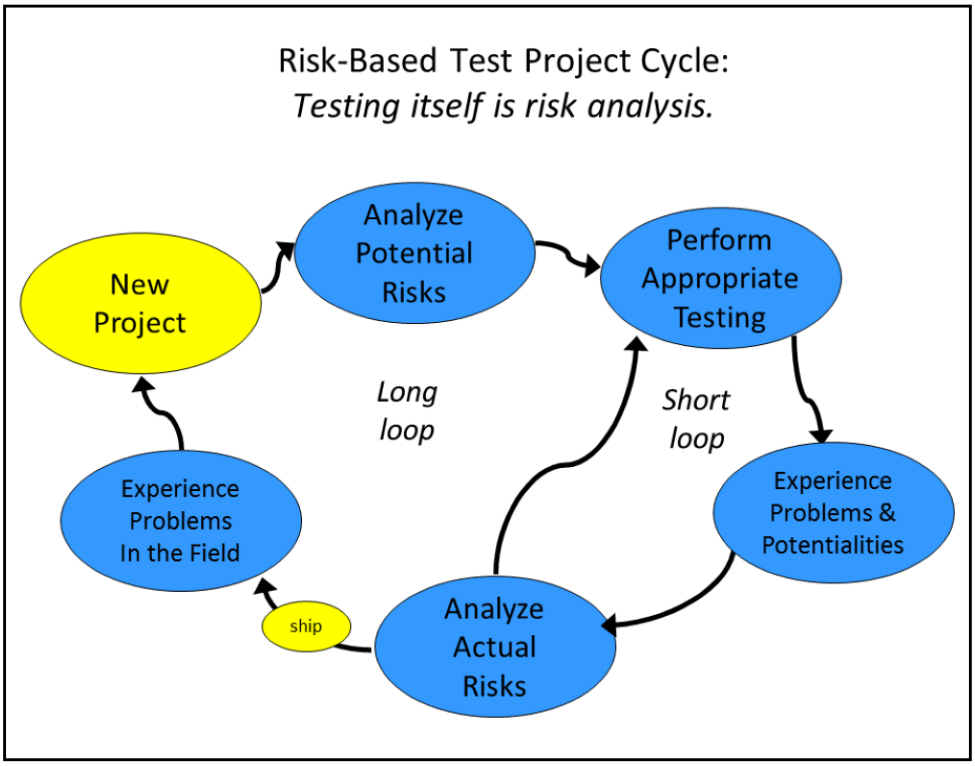
- There is no single technique that finds all bugs.
- We can't do any technique perfectly.
- We can't do all conceivable techniques.

Use **"diverse half-measures"--** lots of different points of view, approaches, techniques, even if no one strategy is performed completely.

Value (or Risk) as a Simplifying Factor
Find problems that matter

- In general it can vastly simplify testing if we focus on whether the product has a problem that matters, rather than whether the product merely satisfies all relevant standards.
- Effective testing requires that we understand standards as they relate to how our clients value the product.

Instead of thinking pass vs. fail,
consider thinking problem vs. no problem.



Testing is itself a form of risk analysis: risk-based testing begins with a rumor of risk and transforms that into information.

Even if testing reveals a problem that cannot be reproduced, you have learned something about risk.

Exploratory testing is a popular and powerful approach to analyzing risk. *Scripted* testing is better suited for monitoring known risk.

What is testing?

Serving Your Client

If you don't have an understanding and an agreement on what is the mission of your testing, then doing it "rapidly" would be pointless.

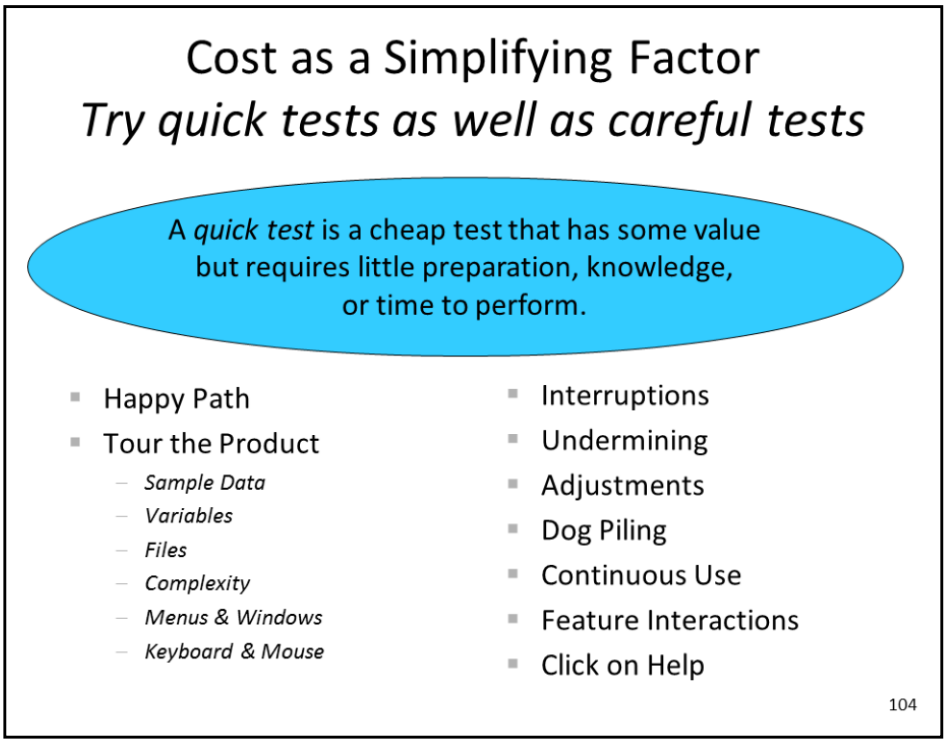
Know your mission.

↘

Begin sympathetically...

↘

...Then chase the risk!



Happy Path: Use the product in the most simple, expected, straightforward way, just as the most optimistic programmer might imagine users to behave. Perform a task, from start to finish, that an end-user might be expected to do. Look for anything that might confuse, delay, or irritate a reasonable person.

Documentation Tour: Look in the online help or user manual and find some instructions about how to perform some interesting activity. Do those actions. Improvise from them. If your product has a tutorial, follow it. You may expose a problem in the product or in the documentation; either way, you've found something useful. Even if you don't expose a problem, you'll still be learning about the product.

Sample Data Tour: Employ any sample data you can, and all that you can—the more complex or extreme the better. Use zeroes where large numbers are expected; use negative numbers where positive numbers are expected; use huge numbers where modestly-sized ones are expected; and use letters in every place that's supposed to handle numbers. Change the units or formats in which data can be entered. Challenge the assumption that the programmers have thought to reject inappropriate data.

Variables Tour: Tour a product looking for anything that is variable and vary it. Vary it as far as possible, in every dimension possible. Identifying and exploring variations is part of the basic structure of my testing when I first encounter a product.

Complexity Tour: Tour a product looking for the most complex features and using challenging data sets. Look for nooks and crowds where bugs can hide.

File Tour: Have a look at the folder where the program's .EXE file is found. Check out the directory structure, including subs. Look for READMEs, help files, log files, installation scripts, .cfg, .ini, .rc files. Look at the names of .DLLs, and extrapolate on the functions that they might contain or the ways in which their absence might undermine the application.

Menus and Windows Tour: Tour a product looking for all the menus (main and context menus), menu items, windows, toolbars, icons, and other controls.

Keyboard and Mouse Tour: Tour a product looking for all the things you can do with a keyboard and mouse. Run through all of the keys on the keyboard. Hit all the F-keys. Hit Enter, Tab, Escape, Backspace. Run through the alphabet in order. Combine each key with Shift, Ctrl, and Alt. Also, click on everything.

Interruptions: Start activities and stop them in the middle. Stop them at awkward times. Perform stoppages using cancel buttons, O/S level interrupts (ctrl-alt-delete or task manager), arrange for other programs to interrupt (such as screensavers or virus checkers). Also try suspending an activity and returning later.

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

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

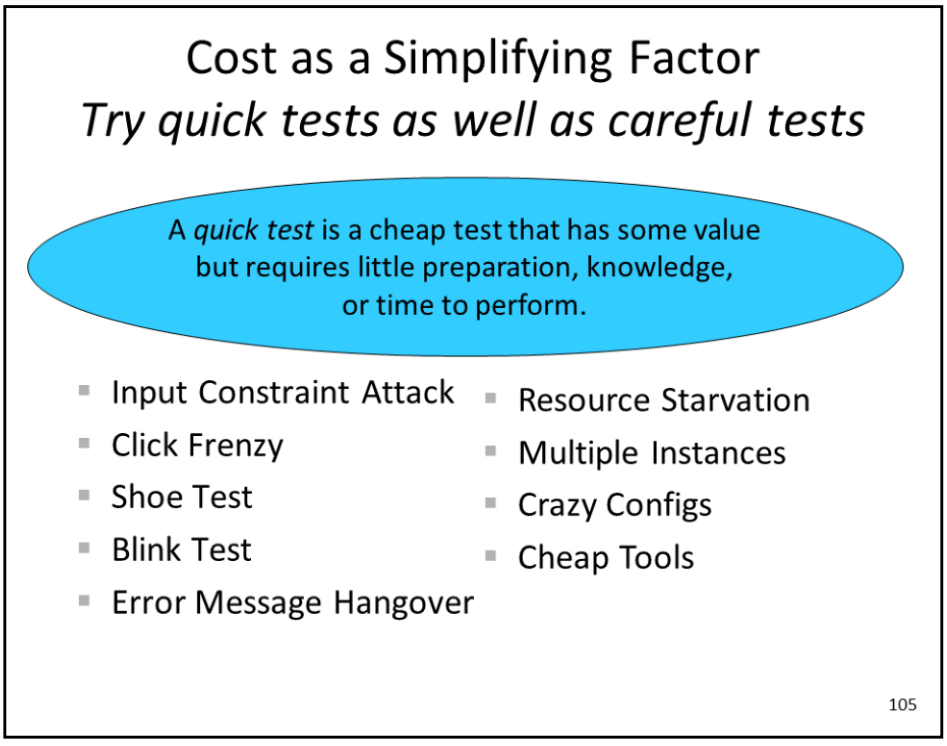
Dog Piling: Get more processes going at once; more states existing concurrently. Nested dialog boxes and non-modal dialogs provide opportunities to do this.

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

Feature Interactions: Discover where individual functions interact or share data. Look for any interdependencies. Tour them. Stress them. I once crashed an app by loading up all the fields in a form to their maximums and then traversing to the report generator. Look for places where the program repeats itself or allows you to do the same thing in different places.

Click for Help: At some point, some users are going to try to bring up the context-sensitive help feature during some operation or activity. Does

the product's help file explain things in a useful way, or does it offend the user's intelligence by simply restating what's already on the screen? Is help even available at all?



Input Constraint Attack: Discover sources of input and attempt to violate constraints on that input. For instance, use a geometrically expanding string in a field. Keep doubling its length until the product crashes. Use special characters. Inject noise of any kind into a system and see what happens. Use Satisfice's PerlClip utility to create strings of arbitrary length and content; use PerlClip's counterstring feature to create a string that tells you its own length so that you can see where an application cuts off input.

Click Frenzy: Ever notice how a cat or a kid can crash a system with ease? Testing is more than "banging on the keyboard", but that phrase wasn't coined for nothing. Try banging on the keyboard. Try clicking everywhere. I broke into a touchscreen system once by poking every square centimeter of every screen until I found a secret button.

Shoe Test: This is any test consistent with placing a shoe on the keyboard. Basically, it means using auto-repeat on the keyboard for a very cheap stress test. Look for dialog boxes so constructed that pressing a key leads to, say, another dialog box (perhaps an error message) that also has a button connected to the same key that returns to the first dialog box. That way you can place a shoe (or Coke can, as I often do, but sweeping off a cowboy boot has a certain drama to it) on the keyboard and walk away. Let the test run for an hour. If there's a resource or memory leak, this kind of test will expose it.

Blink Test: Find some aspect of the product that produces huge amounts of data or does some operation very quickly. For instance, look a long log file or browse database records very quickly. Let the data go by too quickly to see in detail, but notice trends in length or look or shape of the data. Some bugs are easy to see this way that are hard to see with detailed analysis. Use Excel's conditional formatting feature to highlight interesting distinctions between cells of data.

Error Message Hangover: Make error messages happen and test hard after they are dismissed. Often developers handle errors poorly.

Resource Starvation: Progressively lower memory, disk space, display resolution, and other resources until the product collapses, or gracefully (we hope) degrades.

Multiple Instances: Run a lot of instances of the app at the same time. Open the same files. Manipulate them from different windows.

Crazy Configs: Modify the operating system's configuration in non-standard or non-default ways either before or after installing the product. Turn on "high contrast" accessibility mode, or change the localization defaults. Change the letter of the system hard drive. Consider that the product has configuration options, too—change them or corrupt them in a way that should trigger an error message or an appropriate default behavior.

Cheap Tools: Learn how to use InCtrl5, Filemon, Regmon, AppVerifier, Perfmon, and Process Explorer, and Task Manager (all of which are free). Have these tools on a thumb drive and carry it around. Also, carry a digital camera. I now carry a tiny 3 megapixel camera and a tiny video camera. Both fit into my coat pockets. I use them to record screen shots and product behaviors. While it's not cheap, you can usually find Excel on most Windows systems; use it to create test matrices, tables of test data, charts that display performance results, and so on. Use the World-Wide Web Consortium's HTML Validator at http://validator.w3c.org. Pay special attention to tools that hackers use; these tools can be used for good as well as for evil. Netcat, Burp Proxy, wget, and fuzzer are but a few examples.

Can tests be repeated?
Only partly…

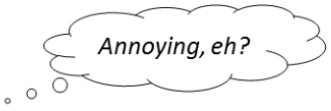
- You can't be certain that you control all the factors that might matter to fulfill the purpose of a test.
- So, to "repeat a test" means that you believe you are repeating *some part of a test that matters*, while other parts of that test may not be repeated.
- Even if you repeat "just 1% of a test", it may be fair to say that you have repeated that test *in every way that matters.*

Exact repetition is not an option.

Minefield Argument

Should tests be repeated?
Think twice before repeating yourself…

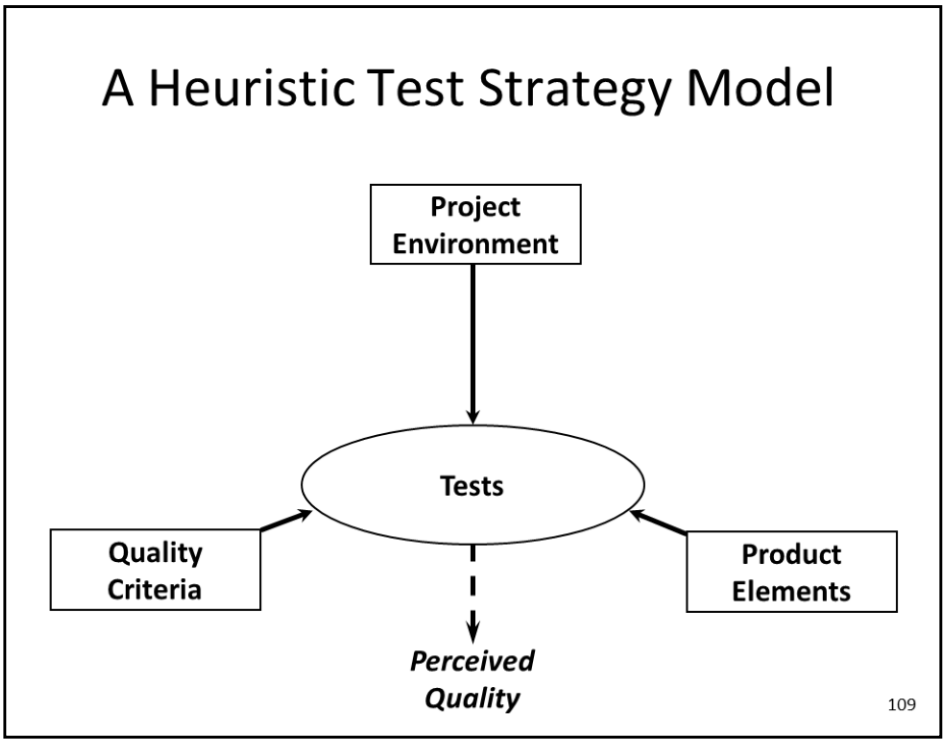
- Consider cost vs. value.
- Consider what could be gained from some of the many tests you have not yet *ever* performed.
- To test is to question:
  - Why ask the same question again?
  - Why ask the same question again?
  - Why ask the same question again?
  - Why ask the same question again?

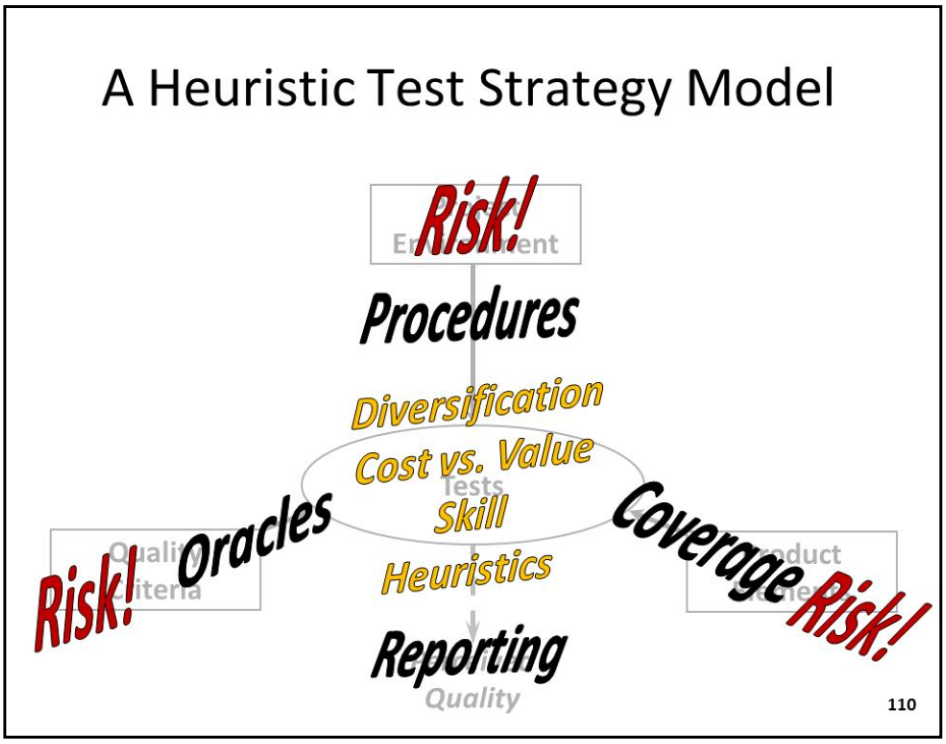


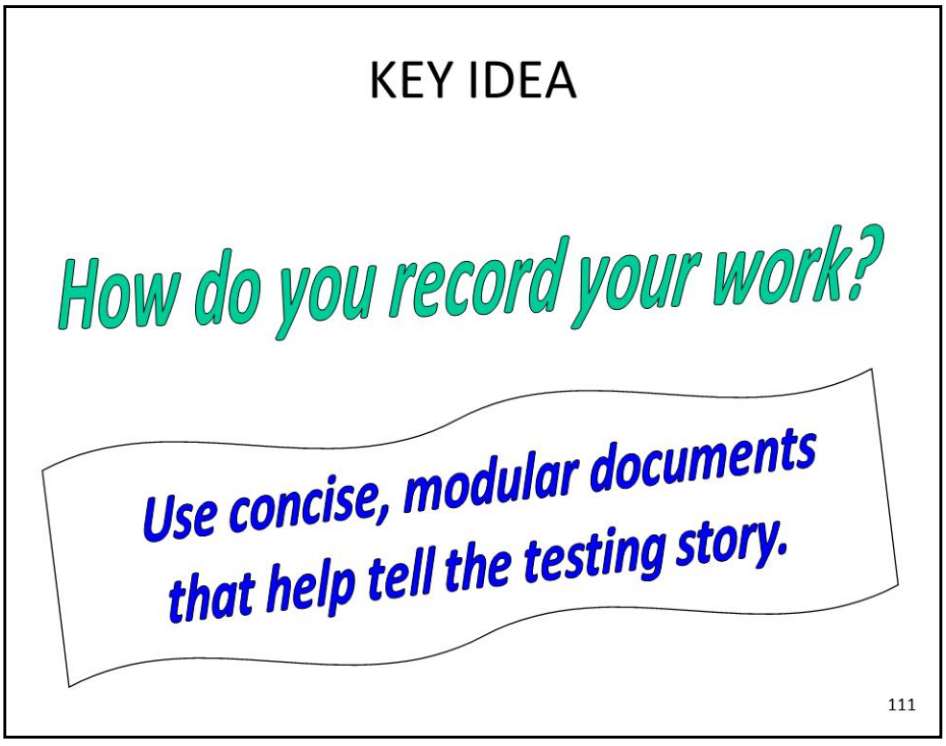
- You should know why, and why *not*, to repeat tests.

Should Tests Be Repeated?
Exploiting Variation To Find More Bugs

- **Micro-behaviors:** Unreliable and distractible humans make each test a little bit new each time through.
- **Randomness:** Can protect you from unconscious bias.
- **Data Substitution:** The same actions may have dramatically different results when tried on a different database, or with different input.
- **Platform Substitution:** Supposedly equivalent platforms may not be.
- **Timing/Concurrency Variations:** The same actions may have different results depending on the time frame in which they occur and other concurrent events.
- **Scenario Variation:** The same functions may operate differently when employed in a different flow or context.
- **State Pollution:** Hidden variables of all kinds frequently exert influence in a complex system. By varying the order, magnitude, and types of actions, we may accelerate state pollution, and discover otherwise rare bugs.
- **Sensitivities and Expectations:** Different testers may be sensitive to different factors, or make different observations. The same tester may see different things at different times or when intentionally shifting focus to different things.



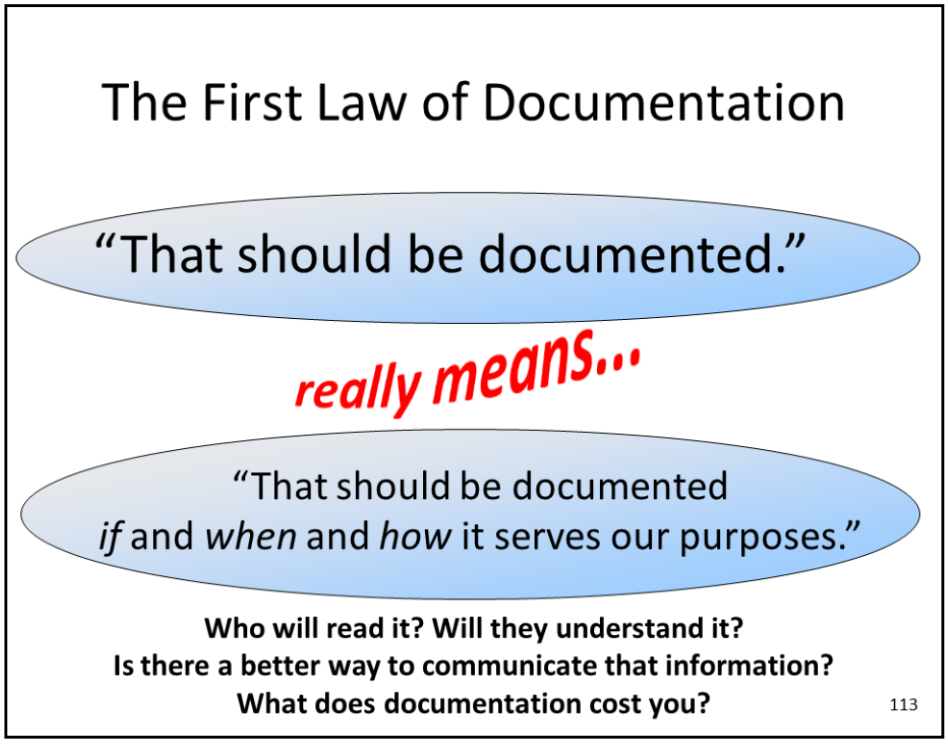




Common Claims About Test Documentation

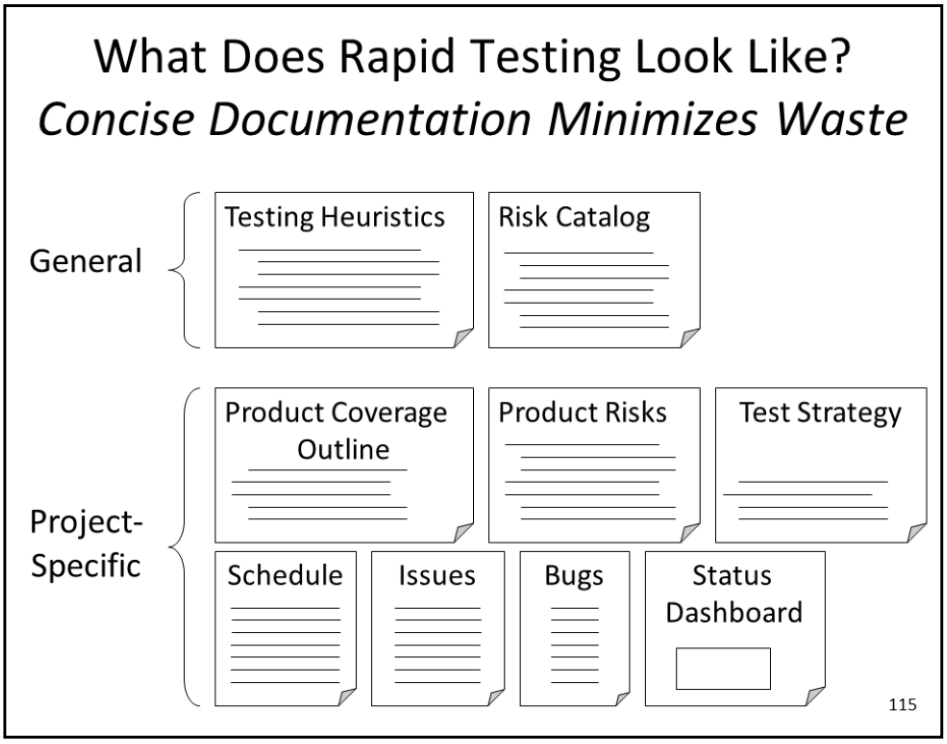
- It's needed for new testers.
- It's needed to deflect legal liability.
- It's needed to share testing with other testers.
- It's needed for "repeatability."
- It's needed for accountability.
- It forces you to think about test strategy.

Something is wrong with these claims.
Can you see the problem?



Common Problems with Test Documentation

- Distracts, and even prevents, testers from doing the best testing they know how to do.
- Authors don't understand testing.
- Authors don't own format.
- Templates help hide poor thinking.
- Full of fluff.
- Fluff discourages readers and increases cost of maintenance.
- No one knows documentation requirements.
- Too much formatting increases the cost of maintenance.
- Information for different audiences is mixed together.
- Catering to rare audiences instead of probable user.
- Disrupts the social life of information.
- Long term and short term goals conflict.
- Most people don't read.



Detailed procedural documentation is expensive and largely unnecessary.

Tutorial documentation is also usually unnecessary, but if you do it, then keep it separate from the working documents.

Consider Automatic Logging

- Exploratory testing works better when the product produces an automatic log of everything that was covered in each test.
- You can also use external logging tools such as Spector (www.spectorsoft.com).
- Automatic logging means that you get something like a retrospective script of what was tested.

117

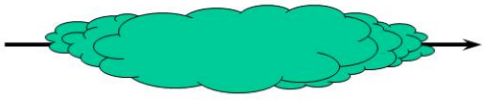
Taking Notes

- Coverage
- Oracles
- Procedures (key elements only)
- Test Ideas
- Bugs/Risks

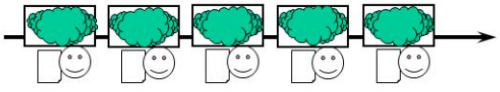
- Issues, Questions & Anomalies
  - It would be easier to test if you changed/added…
  - How does … work?
  - Is this important to test? How should I test it?
  - I saw something strange…

High Accountability Exploratory Testing

1) Charter
2) Time Box
3) Reviewable Result
4) Debriefing



VS.



119

Charter:

A clear mission for the session

- A charter may suggest what should be tested, how it should be tested, and what problems to look for.
- A charter is not meant to be a detailed plan.
- General charters may be necessary at first:
  - "Analyze the Insert Picture function"
- Specific charters provide better focus, but take more effort to design:
  - "Test clip art insertion. Focus on stress and flow techniques, and make sure to insert into a variety of documents. We're concerned about resource leaks or anything else that might degrade performance over time."

Time Box:

Focused test effort of fixed duration

Short: 60 minutes (+-15)
Normal: 90 minutes (+-15)
Long: 120 minutes (+-15)

- Brief enough for accurate reporting.
- Brief enough to allow flexible scheduling.
- Brief enough to allow course correction.
- Long enough to get solid testing done.
- Long enough for efficient debriefings.
- Beware of overly precise timing.

Debriefing:

Measurement begins with observation

- The manager reviews ***session sheet*** to assure that he understands it and that it follows the protocol.
- The tester answers any questions.
- Session metrics are checked.
- Charter may be adjusted.
- Session may be extended.
- New sessions may be chartered.
- Coaching happens.

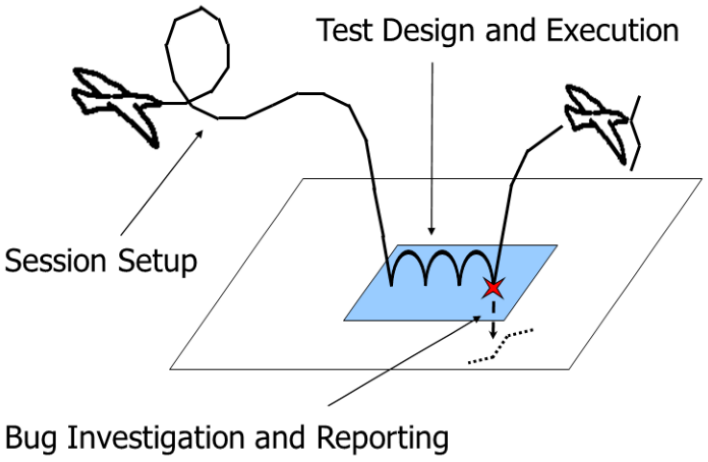
Reviewable Result:
A scannable session sheet

- Charter
  - #AREAS
- Start Time
- Tester Name(s)
- Breakdown
  - #DURATION
  - #TEST DESIGN AND EXECUTION
  - #BUG INVESTIGATION AND REPORTING
  - #SESSION SETUP
  - #CHARTER/OPPORTUNITY
- Data Files
- Test Notes
- Bugs
  - #BUG
- Issues
  - #ISSUE

123

The Breakdown Metrics

Testing is like looking for worms



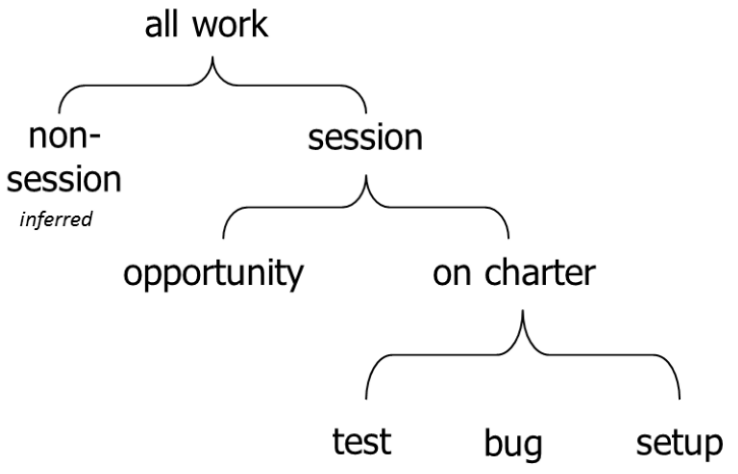
Reporting the TBS Breakdown

A guess is okay, but follow the protocol

- Test, Bug, and Setup are orthogonal categories.
- Estimate the percentage of charter work that fell into each category.
- Nearest 5% or 10% is good enough.
- If activities are done simultaneously, report the highest precedence activity.
- Precedence goes in order: T, B, then S.
- All we really want is to track interruptions to testing.
- Don't include Opportunity Testing in the estimate.

Activity Hierarchy

All test work fits here, somewhere

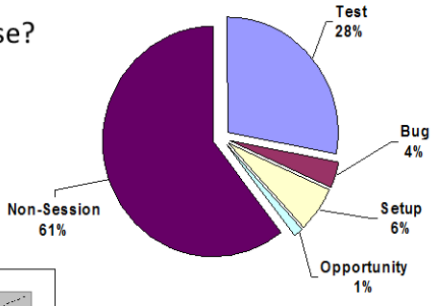


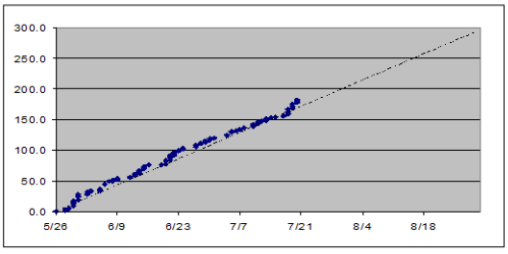
126

Work Breakdown:

Diagnosing the productivity

- Do these proportions make sense?
- How do they change over time?
- Is the reporting protocol being followed?





Coverage:

Specifying coverage areas

- These are text labels listed in the Charter section of the session sheet. (e.g. “insert picture”)
- Coverage areas can include anything
  - areas of the product
  - test configuration
  - test strategies
  - system configuration parameters
- Use the debriefings to check the validity of the specified coverage areas.

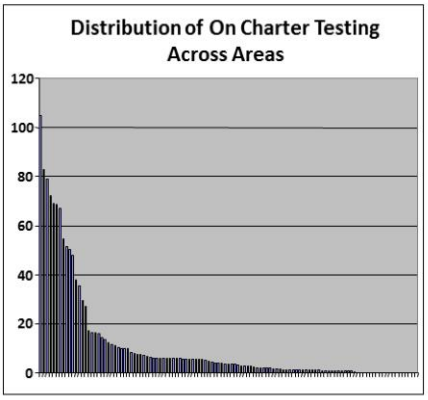
Coverage:

Are we testing the right stuff?

- Is this a risk-based test strategy?

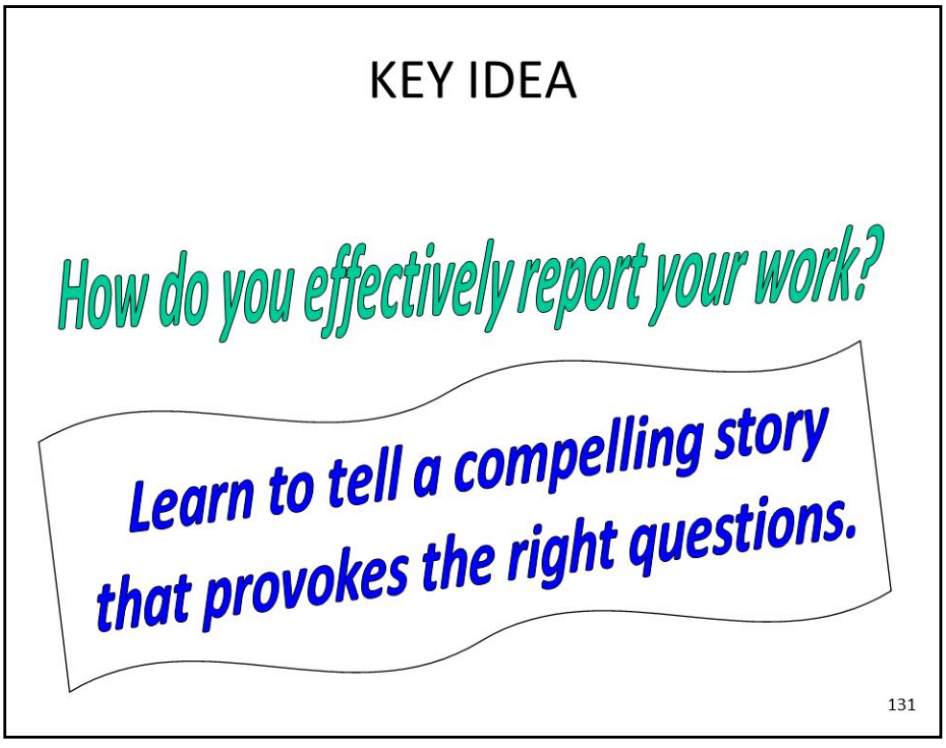
or

- Is it a lop-sided set of coverage areas?
- Is it distorted reporting?



Using the Data to Estimate a Test Cycle

1. How many perfect sessions (100% on-charter testing) does it take to do a cycle? *(let's say 40)*
2. How many sessions can the team (of 4 testers) do per day? *(let's say 3 per day, per tester = 12)*
3. How productive are the sessions? *(let's say 66% is on-charter test design and execution)*
4. Estimate: 40 / (12 * .66) = **5 days**
5. We base the estimate on the data we've collected. When any conditions or assumptions behind this estimate change, we will update the estimate.



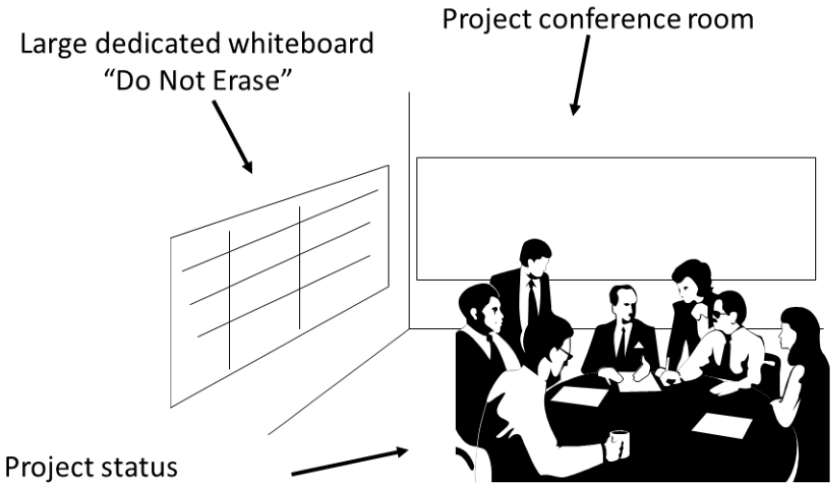
0 to 100

Reporting Considerations

- **Reporter safety:** What will they think if I made no progress?
- **Client:** Who am I reporting to and how do I relate to them?
- **Rules:** What rules and traditions are there for reporting, here?
- **Significance of report:** How will my report influence events?
- **Subject of report:** On what am I reporting?
- **Other agents reporting:** How do other reports affect mine?
- **Medium:** How will my report be seen, heard, and touched?
- **Precision and confidence levels:** What distinctions make a difference?

Take responsibility for the communication.

The Dashboard Concept



Test Cycle Report

Product Areas

vs.

Test Effort

Test Coverage

Quality Assessment

vs.

Time

134

Testing Dashboard				Updated: 2/21 Build: 38
Area	**Effort**	**C.**	**Q.**	**Comments**
file/edit	high	1	☺	
view	low	1+	😐	1345, 1363, 1401
insert	low	2	☺	
format	low	2+	😐	automation broken
tools	blocked	1	☹	crashes: 1406, 1407
slideshow	low	2	☹	animation memory leak
online help	blocked	0		new files not delivered
clipart	none	1	😐	need help to test...
converters	none	1	😐	need help to test...
install	start 3/17	0		
compatibility	start 3/17	0		lab time is scheduled
general GUI	low	3	☺	

135

Product Area

Area
file/edit
view
insert
format
tools
slideshow
online help
clipart
converters
install
compatibility
general GUI

- 15-30 areas (keep it simple)
- Avoid sub-areas: they're confusing.
- Areas should have roughly equal value.
- Areas together should be inclusive of everything reasonably testable.
- "Product areas" can include tasks or risks- but put them at the end.
- Minimize overlap between areas.
- Areas must "make sense" to your clients, or they won't use the board.

Test Effort

None	Not testing; not planning to test.
Start	No testing yet, but expect to start soon.
Low	Regression or spot testing only; maintaining coverage.
High	Focused testing effort; increasing coverage.
Pause	Temporarily ceased testing, though area is testable.
Blocked	Can't effectively test, due to blocking problem.
Ship	Going through final tests and signoff procedure.

Test Effort

- Use red to denote significant problems or stoppages, as in blocked, none, or pause.
- Color ship green once the final tests are complete and everything else on that row is green.
- Use neutral color (such as black or blue, but pick only one) for others, as in start, low, or high.

Test Coverage

0	We don't have good information about this area.	
1	Sanity Check:	major functions & simple data.
1+	More than sanity, but many functions not tested.	
2	Common & Critical:	all functions touched; common & critical tests executed.
2+	Some data, state, or error coverage beyond level 2.	
3	Complex Cases:	strong data, state, error, or stress testing.

139

Test Coverage

- Color green if coverage level is acceptable for ship, otherwise color black.
- Level 1 and 2 focus on functional requirements and capabilities: *can* this product work at all?
- Level 2 finds any "obvious" bug.
- Level 2+ and 3 focus on information to judge performance, reliability, compatibility, and other "ilities": *will* this product work under realistic usage?
- Level 3 or 3+ implies "if there were a bad bug in this area, we would probably know about it."

Quality Assessment

"We know of no problems in this area that threaten to stop ship or interrupt testing, nor do we have any definite suspicions about any."

"We know of problems that are possible showstoppers, or we suspect that there are important problems not yet discovered."

"We know of problems in this area that definitely stop ship or interrupt testing."

Comments

Use the comment field to explain anything colored red, or any non-green quality indicator.

- Problem ID numbers.
- Reasons for pausing, or delayed start.
- Nature of blocking problems.
- Why area is unstaffed.

Using the Dashboard

- **Updates:** 2-5/week, or at each build, or prior to each project meeting.
- **Progress:** Set expectation about the duration of the "Testing Clock" and how new builds reset it.
- **Justification:** Be ready to justify the contents of any cell in the dashboard. The authority of the board depends upon meaningful, actionable content.
- **Going High Tech:** Sure, you can put this on the web, but will anyone actually look at it???

KEY IDEA

Rapid testing is a personal discipline. It requires practice, but no one's permission.

144

The Themes of Rapid Testing

- Put the **tester's mind** at the center of testing.
- Learn to **deal with complexity** and ambiguity.
- Learn to **tell a compelling testing story.**
- Develop **testing skills** through practice, not just talk.
- **Use heuristics** to guide and structure your process.
- **Be a service** to the project community, not an obstacle.
- **Consider cost vs. value** in all your testing activity.
- **Diversify** your team and your tactics.
- **Understand your users**, don't just read specs.
- Dynamically **manage the focus** of your work.
- Your **context should drive your choices**, both of which evolve over time.

Getting started…

- **Take advantage of resources**
  - Email me
  - See my book Lessons Learned in Software Testing
  - Browse the appendices of the class
  - Join forum: software-testing@yahoogroups.com
- **Do some focused exploration**
  - Try a three-hour chartered bug hunt with a group of testers. *See what happens*.
  - Watch yourself as you learn a new product. Notice how your thoughts evolve.
- **Practice solving puzzles**
  - Solve jigsaw puzzles, logic puzzles, Sudoku, cryptograms, or lateral thinking puzzles.

Getting started…

- **Try using guideword heuristics**
  - Use my Heuristic Test Strategy Model
  - Or modify my model
  - Or make your own model
- **Defrost your procedures**
  - Pick some procedures and simplify them.
  - Generalize some of your procedures.
  - Include design information to help test executors contribute.
- **Start a lunch-time test coaching meeting**
  - Pick a part of the product each week and talk about how to test it. Get the whole team to brainstorm better ways.
  - Do a risk brainstorm.
  - Review bug reports and talk about how to write better ones.

Getting started…

- **Practice your systems thinking**
  - Read "The Art of Systems Thinking" or "An Introduction to General Systems Thinking"
  - Try the exercises in those books.
  - Take any part of your product and model it in terms of its most important dimensions.
- **Put a coverage/risk outline inside your test documentation**
  - You don't have to throw out your old test documentation, but how about putting a new section in that consists of a coverage outline, a risk outline, or a combination of the two?
- **Question value vs. cost for all your testing**
  - Is any of your testing not giving you much value relative to its cost? Why are you doing it, then?