Test Framing

To test is to tell two parallel stories: a *story of the product*, and the *story of our testing*. Test framing is a key skill that helps us to compose, edit, narrate, and justify the story of our testing in a logical, coherent, and rapid way. The goal of test framing is to link each testing activity with the testing mission.

Elements of Test Framing

The basic idea is this: in any given testing situation

- You have a testing mission, a search for information. Your mission may change over time.
- You have information about requirements. Some of that information is explicit, some implicit; and it will likely change over time.
- You have risks that inform the mission. Awareness and priority of those risks will change over time.
- You have ideas about what would provide value in the product, and what would threaten value. You'll refine those ideas as you test.
- You have a context in which you're working. That context will change over time.
- You will apply oracles, principles or mechanisms by which you would recognize a problem. You will refine some oracles and discover others as you go.
- You have models of the product that you intend to cover. You will refine and extend those models throughout the project.
- You have test techniques that you may apply. You also have choices about which techniques you use, and how you apply them. You will develop techniques along the way.
- You have lab procedures that you follow. You may wish to follow them more or less strictly from time to time.
- You have skills and heuristics that you may apply. Those skills and heuristics will develop on this project, and in your overall career as a tester.
- You have issues related to the cost versus the value of your activities that you must assess.
- You have tests to perform. These are usually fewer than the tests you would like to perform. In any case, you choose your tests from infinite number of possible tests.
- You have time in which to perform your tests. Your available time is probably severely limited compared to the time needed for tests you'd like to perform. The time available to perform tests asymptotically approaches zero, relative to the time required to perform all possible tests.

Test framing is the capacity to follow and express, at any time, a direct line of logic that connects the mission to the tests. Such a line of logical reasoning will typically touch on elements between the top and the bottom of the list above.

Purpose of Test Framing

The *purpose* of test framing is to be able to provide clear, logical, credible answers to questions like

- Why are you running (did you run, will you run) this test (and not some other test)?
- Why are you running that test now (did you run that test then, will you run that test later)?
- Why are you testing (did you test, will you test) for this requirement, rather than that requirement?
- How are you testing (did you test, well you test) for this requirement?
- How does the configuration you used in your tests relate to the real-world configuration of the product?
- How does your test result relate to your test design?
- Was the mission related to risk? How does this test relate to that risk?
- How does this test relate to other tests you might have chosen?
- Are you qualified (were you qualified, can you become qualified) to test this?
- Why do you think that is (was, would be) a problem?

The Form of Test Framing

The form of test framing is a line of propositions and logical connectives that relate the test to the mission, touching on the elements of testing as listed above.

A *proposition* is a simple statement that expresses a concept. The statement may be true or false. In test framing, we typically use propositions as affirmative declarations or assumptions. Occasionally, we will use propositions as the basis of hypotheses to be tested or falsified.

Connectives are word or phrases that link or relate propositions to each other, generating new propositions by inference. Examples include “and”, “not”, “if”, “therefore”, “and so”, “unless”, “because”, “since”, “on the other hand”, “but maybe”, and so forth.

The kind of language used in test framing need not be a *strictly* formal system, but one that is heuristic and reasonably well structured. Here are a couple of fairly straightforward examples.

Example 1:

Mission: Find problems that might threaten the value of the product, such as program misbehaviour or data loss.

Proposition: There’s an input field here.
Proposition: Upon the user pressing Enter, the input field sends data to a buffer.
Proposition: Unconstrained input may overflow a buffer.
Proposition: Buffers that overflow clobber data or program code.
Proposition: Clobbered data can result in data loss.

Proposition: Clobbered program code can result in observable misbehaviour.

Connecting the propositions: IF this input field is unconstrained, AND IF it consequently overflows a buffer, THEREFORE there's a risk of data loss OR program misbehaviour.

Proposition: The larger the data set that is sent to this input field, the greater the chance of clobbering program code or data.

Connection: THEREFORE, the larger the data set, the better chance of triggering an observable problem.

Connection: IF I put an extremely long string into this field, I'll be more likely to observe the problem.

Conclusion: THEREFORE, as a test, I will try to paste an extremely long string in this input field AND look for signs of mischief such as garbage in records that I observed as intact before, or memory leaks, or crashes, or other odd behaviour.

Example 2

Mission: Evaluate our product and its interaction with a related product.

Proposition: Our product, MobileMoolah, allows users to record financial transactions at the point of sale, and to upload batches of those transactions to a compatible products.
Proposition: PC ChequeBook has the second-largest market share of desktop finance products, where CompuCash is the market leader.
Proposition: PC ChequeBook allows localized date formats (mm/dd/yyyy, dd/mm/yyyy, yyyy-mm-dd, and so forth).
Proposition: MobileMoolah also allows localized date formats.

Connection: If we set the date formats to the same setting in both products, we can reasonably expect to import data from MobileMoolah to PC ChequeBook without having to modify the date manually.

Proposition: A goal of any software product is to save time and increase convenience for a customer.
Proposition: One oracle that would point to a problem in the product is inconsistency with its explicit or implicit purposes.
Proposition: Another oracle that would point to a problem in the product is inconsistency with its reasonable user expectations.

Proposition: Upon importing data from MobileMoolah into PC Chequebook, with both date formats set to dd/mm/yyyy, we observe that the imported transactions contain the format mm/dd/yyyy.

Connection: If, after the transfer of data between PC ChequeBook and MobileMoolah, the month and day fields appear switched for any or all transactions, the product is inconsistent with a reasonable user expectation, and inconsistent with the product's implicit purpose. THEREFORE we have reason to suspect a bug.

Proposition: Based on this test alone, we cannot determine without further investigation whether the bug is in PC ChequeBook or MobileMoolah.

Proposition: The new update to MobileMoolah is slated to ship in four days.

Connection: IF our mission is to find as many bugs as we can before ship time, we should quickly reproduce this bug, and report it immediately (along with the steps to reproduce it) without further investigation.

Connection: IF, at this time, this is the most serious problem that we've found, AND IF we believe our mission includes discovering whether this is a general problem rather than a problem specific to PC ChequeBook, we should reproduce this problem with a different desktop program (probably the market leader) before moving on.

Proposition: We have a CompuCash test system available to us, on which we can run this same test immediately and have a result within two minutes.

Connection: IF the problem were to reproduce with CompuCash, THEREFORE we would be able to infer that we're seeing a general problem with MobileMoolah.

Conclusion: SINCE we can get some more useful information quickly and specific information quickly, we might as well.

Now, to some, the thought process in these examples might sound quite straightforward and logical. However, in our experience, some testers have surprising difficulty with tracing the path from mission down to the test, or from the test back up to mission—or with expressing the line of reasoning immediately and cogently.

Our approach, so far, is to give testers something to test and a mission. We might ask them, given the mission, to describe a test that they might choose to run; and to have them describe their reasoning. As an alternative, we might ask them why they chose to run a particular test, and to explain that choice in terms of tracing a logical path back to the mission.

Unframed Tests

If you have an unframed test, try framing it. You should be able to do that for most of your tests, but if you can't frame a given test right away, it might be okay. Why? Because as we test, we not only apply information; we also *reveal* it. Therefore, we think it's usually a good idea to alternate between focusing and defocusing approaches. After you've been testing very systematically using well-framed tests, mix in some tests that you can't immediately or completely justify.

One of the possible justifications for an unframed test is that **we're always dealing with hidden frames**. Revealing hidden or unknown frames is a motivation behind randomized high-volume automated tests, or stress tests, or galumphing, or any other test that might (but not certainly) reveal a startling result. The fact that you're startled provides a reason, in retrospect, to have performed the test. So, you might justify unframed tests in terms of plausible outcomes or surprises, rather than known theories of error. You might encounter a "predicatable" problem, or one more surprising to you. In that case, better that *you* should say "Who knew?!" than a customer.

Structures for Test Framing

Part of the skill of test framing involves identifying structures that inform testing, and the elements of those structures. The following quick references might be helpful in constructing the testing story.

Testing Mission: Testing is not necessarily a simple matter of finding bugs. There are many possible missions for testing, including informing ship/no-ship decisions, competitive evaluation, or finding safe scenarios and workarounds for problems. “Different objectives require different testing tools and strategies, and will yield different tests, different test documentation, and different results.” See page 19 of Cem Kaner, “Challenges in the Evolution of Software Testing Practices in Mission-Critical Environments.” Software Test & Evaluation Summit/Workshop (National Defense Industrial Association), Reston VA, September 2009. http://www.kaner.com/pdfs/NDIAkanerSept2009.pdf

Information about requirements: On any project, there's always more information available than one might think at first glance. The trick is to be able to find and exploit those sources of information quickly and consciously. Consider *reference*, *inference*, and *conference* as heuristic sources of knowledge. They're all useful, they're all incomplete, and each may contradict, reinforce, or refine the other. See Michael Bolton, “Rock, Paper, Scissors”, *Better Software, Vol. 8, No. 11, December 2006*. http://www.developsense.com/articles/2006-11-RockPaperScissors.pdf

Requirements as reference, conference, and inference is also discussed in Kaner, Bach, and Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach.* Wiley, 2001.

Risks: There’s a four-part story to risk, in which a victim—some person—suffers harm or loss or annoyance, due to a weakness in the program that is triggered by some threat. For a brief introduction to risk ideas, consider James Bach, “Heuristic Risk-Based Testing” in *Software Testing and Quality Engineering, 11/99*. Also look at http://www.satisfice.com/articles/hrbt.pdf Michael Bolton, “Test Design with Risk in Mind”. *Better Software, Vol. 9, No. 7, July 2007*. For a more detailed list, see Kaner, Falk, and Nguyen, *Testing Computer Software, Second Edition,* Wiley 1999. For a discussion of risk in general, see Nassim Nicholas Taleb, *The Black Swan: The Impact of the Highly Improbable (Second Edition)*, Random House Trade Paperbacks, 2010.

Value: Value in a product is always subjective and multi-dimensional. For a quick reference to some of ways people might evaluate a product, see the “Quality Criteria” section of the Heuristic Test Strategy Model, attached to this document. See also Donald Gause and Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

Context: See “The Satisfice Test Context Model” and the Project Environment section of the “Heuristic Test Strategy Model”, attached to this document.

Oracles: See the (previously unpublished) article “Oracles” attached to this document. See also Michael Bolton, “Testing Without A Map”, *Better Software Vol. 7, No. 1, January 2005*. http://www.developsense.com/articles/2005-01-TestingWithoutAMap.pdf

Coverage: The first step when we are seeking to evaluate or enhance the quality of our test coverage is to determine for whom we're determining coverage, and why. A mapping illustrates a relationship between two things. In testing, a map might look like a road map, but it might also look like a list, a chart, a table, or a pile of stories. We can use any of these to help us think about test coverage. Yet excellent testing isn't just about covering the "map"—it's also about exploring the territory, which is the process by which we discover things that the map doesn't cover. See Michael Bolton, "Got You Covered", *Better Software, Vol. 10, No. 8, October 2008,* http://www.developsense.com/articles/2008-10-GotYouCovered.pdf; "Cover or Discover", *Better Software, Vol. 10, No. 9, November 2008*, http://www.developsense.com/articles/2008-11-CoverOrDiscover.pdf; and "A Map By Any Other Name", *Better Software, Vol. 10, No. 10*, December 2008, http://www.developsense.com/articles/2008-11-AMapByAnyOtherName.pdf

Test Techniques: See the "Test Techniques" portion of the Heuristic Test Strategy Model. Also see Lee Copeland, A Practitioner's Guide to Software Test Design, Artech House Publishers, 2003.

Skills: See "Exploratory Testing Skills and Dynamics" attached to this document.

Oracles

An oracle is a heuristic principle or mechanism by which someone recognizes a problem.

If we perceive a problem, it's because an oracle is telling us that there's a problem. Conversely, if we don't see a problem, it's because no oracle is telling us that there is a problem. That doesn't mean that there is no problem, or that there's no oracle for a problem that's there. It simply means that, for whatever reason, we're not applying a principle or mechanism that would identify a problem. We may not be aware of the oracle, or the oracles that we have may be misleading us, or failing to lead us far enough.

Oracles are by their nature heuristic. That is, oracles are fallible and context-dependent. Oracles do not tell us conclusively that there is a problem; rather, they suggest that there *may be* a problem. As Gause and Weinberg define it[1], a problem is "a difference between things as perceived and things as desired". Perception and desire are both human and subjective, relative to some situation at some time. Consequently, there can be no absolute oracle. Most testers are familiar with applying an oracle which seems to indicate a problem, whereupon a program manager replies, "That's not a bug; that's a feature." The tester and the program manager here are applying different oracles. Neither is wrong; each is using a different principle or mechanism.

Consistency is an important theme in oracles. Unless there is a compelling reason to desire otherwise, we generally tend to want a product to be consistent with

- **History:** That is, we expect the present version of the system to be consistent with past versions of it. Naturally, if a product is inconsistent with its history because a bug has been fixed, we likely appreciate the inconsistency. Yet if our programmers have provided a workaround for the old problem, and the fix requires us to change our work habits, we may resent the fix! This underscores the point that all oracles are heuristic. Oracles may give inconsistent indications, and they contradict each other. Oracles should be applied thoughtfully, rather than followed.
- **Image:** We expect the system to be consistent with an image that the organization wants to project, with its brand, or with its reputation. This can work both ways; for example, a game producer might specialize in strategy games such that the strategy aspect is paramount and graphic design is relatively unimportant. For such a company, problems with graphics receive less attention than problems with the strategic aspect of the game.
- **Comparable Products:** We expect the system to be consistent with systems that are in some way comparable. That might include other products in the same product line, or from the same company. The consistency-with-past-versions (History) heuristic is arguably a special case of this more general heuristic. Competitive products, services, or systems may be comparable in dimensions that could help to discover a problem. Products that are not in the same category but which process the same data (as a word processor might use the contents of a database for a mail merge) are comparable for the purposes of this heuristic. A paper form is comparable with a computerized input form designed to replace it. Indeed, any

[1] Donald Gause and Gerald M. Weinberg, *Are Your Lights On?* Dorset House Publishing Company, Inc. (March 1, 1990).

product with any feature may provide some kind of basis for comparison, whereby someone might recognize a problem or a suggestion for improvement.

- **Claims:** We expect the system to be consistent with what important people say about it. These claims may take the form of reference (documents or products that you can point to), inference (what you believe someone important might say about the system), or conference (what someone important *does* say). The claim may be incomplete or in error, in which case testing may reveal a problem with the claim, rather than a problem with the system. Important people might disagree in their claims about what the product should do. The tester's role is not to decide the matter, but to make people aware of the disagreement.
- **Users' Expectations:** We expect the system to be consistent with some idea about what its users might want. Consider "users" broadly here. A system that will be used in many different ways will have diverse users whose expectations and desires may conflict. Often the direct user of a product is acting as a proxy for the person who receives the bulk of the benefit of the product or service, as a travel agent is operating a reservation system largely on behalf of her client.
- **Product:** We expect each element of the system to be consistent with comparable elements in the same system. A product might afford several means of accessing or observing a particular variable; consider the different ways of setting the margins—via a visible ruler or via a dialog box—in a word processing program, or differences between screen and print output. User interface elements should be broadly consistent with one another, both for consistency of user interaction and consistency of image.
- **Purpose:** We expect the system to be consistent with the explicit and implicit ways in which people might use it. If some aspect of the product is missing, such that it fails to fulfill the user's needs or support the user's task, we suspect a problem. If the product over-delivers, presenting options or features that confuse, overwhelm, or slow down a user, we suspect a problem.
- **Standards and Statutes:** We expect a system to be consistent with relevant standards or applicable laws. Note that compliance with a standard may be voluntary; a development group may choice to violate a point in a standard or may reject the standard entirely. Yet non-adherence to a standard should be conscious, rather than compulsive. It may be the tester's role to draw attention to non-conformance with relevant standards—or unnecessary conformance with irrelevant standards.

There is one more heuristic that testers commonly apply. Unlike the preceding ones, this one is an *inconsistency* heuristic:

- **Familiarity:** We expect the system to be *inconsistent* with any patterns of familiar problems. Note that any pattern of familiar problems must eventually reduce to one of the eight consistency heuristics.

We can carry this list of consistency heuristics in our heads more easily by applying a mnemonic, based on the first letter of each heuristic guideword: HICCUPPS (F).

These consistency heuristics are subject to Joel Spolsky's Law of Leaky Abstractions ("All non-trivial abstractions are to some extent leaky.") This means that there may be overlap between the heuristics. That's fine; the object is to prevent an important problem, or class of problems, from being overlooked by defining our categories too narrowly.

Since oracles are fallible and context-dependent, testers cannot know the deep truth about any observation or test result. No single oracle can tell you whether a program (or a feature) is working correctly at all times and in all circumstances, so it's important to use a variety of oracles, and to be open to applying new ones at any moment. Any program that looks like it's working may in fact be failing in some way that happens to fool all of your oracles. To defend against that, you must proceed with humility and critical thinking[2].

Because oracles are not fallible, a tester reports whatever seems plausibly to be a problem. How does one decide on plausibility? Testers apply abductive inference, cycles of reasoning to the best explanation. We collect data and observations, we hypothesize explanations to account for the data, and we evaluate the hypotheses. Then we make a decision: choose the hypothesis that best accounts for the data, and stop; or collect more data—or more hypotheses. This too is a heuristic process, and "heuristic devices don't tell you when to stop"[3].

Oracles can be used prospectively or generatively; in the moment that they're applied; or retrospectively. We usually have a large number of oracles at our disposal before we start testing. Yet we often do not have oracles that establish a definite correct or incorrect result in advance.

- You may use an oracle to help design a test. ("If the data doesn't get transmitted to the server after I press update, that would be inconsistent with an implicit purpose and an explicit claim, so I'll look for a problem like that.") Cycle through the list of oracle heuristics while you're engaged in test design.
- You may suddenly become conscious of an oracle ("Hey… that account balance is *negative*! I didn't expect *that* to happen! That's inconsistent with what I *would have* expected, had I anticipated that in advance.")
- You may apply an oracle retrospectively. ("Since that particular standard came to my attention, I realize now that what I saw when I was testing two weeks ago was non-standard behaviour. I'm going to investigate that now that I have a reason to suspect it was a bug.")

At any time subsequent to the test, you may cite an oracle heuristic to explain why you believe something to be a problem. A problem (or non-problem) may be more easily recognized with the application of multiple oracles that agree with each other. Oracles may contradict one another. A product owner's decision on what to do about a problem report may be influenced by choices about which oracles to apply. Therefore, since our role as testers is to provide credible information, we may also choose to use different oracles to temper our test framing or our bug advocacy[4].

For more on oracles, see Cem Kaner, "Introduction: The strategy problem and the oracle problem", *Black Box Software Testing*, Center for Testing Education and Research, Florida Institute of Technology. http://www.testingeducation.org/BBST/BBSTIntro1.html

---

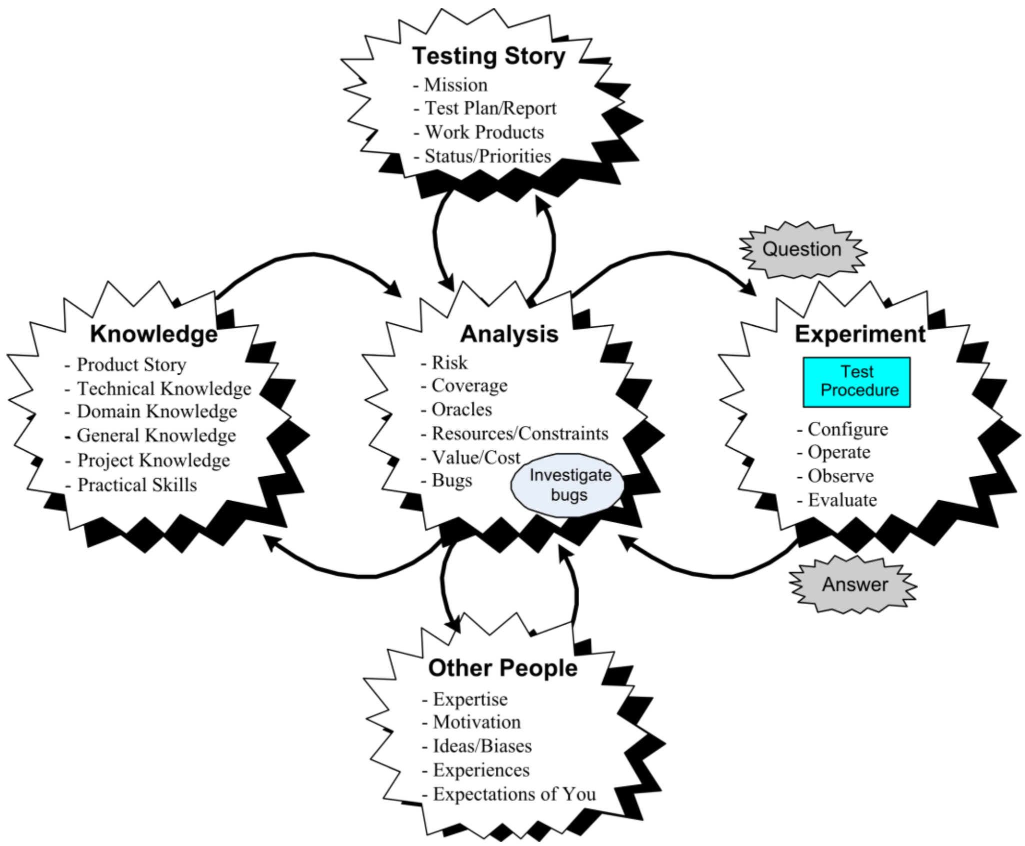
[2] For an excellent introduction, see David Levy, *Tools of Critical Thinking: Metathoughts for Psychology (Second Edition)*. Waveland Press, 2009.
[3] Gerald M. Weinberg, *An Introduction to General Systems Thinking, Silver Anniversary Edition*. Dorset House, 2001.
[4] Cem Kaner and James Bach, "# Bug advocacy: How to win friends, influence programmers, and stomp bugs", *Black Box Software Testing*. Center for Testing Education and Research, Florida Institute of Technology. http://www.testingeducation.org/BBST/BBSTbugAdvocacy.htm

Exploratory Testing Dynamics

Created by James Bach, Jonathan Bach, and Michael Bolton[1] *v3.0*

Exploratory testing is the opposite of *scripted* testing. Both scripted and exploratory testing are better thought of as test *approaches*, rather than techniques. This is because virtually any test technique can be performed in either a scripted or exploratory fashion. Exploratory testing is often considered mysterious and unstructured. Not so! You just need to know what to look for.

The diagram below shows the main elements of exploratory testing modeled as a set of cycles:



In any competent process of testing that is done in an exploratory way, you can expect to find these elements. The arrows represent dynamic influences of the elements on each other, mediated by various forms of thinking. For instance:

Learning: The cycle between analysis and knowledge might be called the learning loop. In this interaction the tester is reviewing and thinking about, and applying what he knows.

Testing: The cycle between analysis and experiment might be called the testing loop. It is dominated by questions which guide the gathering of evidence about the product.

Collaboration: The cycle between analysis and other people might be called the collaboration loop. Collaboration is not necessarily a part of exploration, but often is, especially in larger projects.

Self-management: The cycle between analysis and the testing story is self-management, by which the whole process is regulated.

---

[1] The participants in the Exploratory Testing Research Summit #1 also reviewed this document. They included: James Bach, Jonathan Bach, Mike Kelly, Cem Kaner, Michael Bolton, James Lyndsay, Elisabeth Hendrickson, Jonathan Kohl, Robert Sabourin, and Scott Barber

Evolving Work Products

Exploratory testing spirals upward toward a complete and professional set of test artifacts. Look for any of the following to be created, refined, and possibly documented during the process.

	Test Ideas. Tests, test cases, test procedures, or fragments thereof.
	Testability Ideas. How can the product be made easier to test?
	Test Results. We may need to maintain or update test results as a baseline or historical record.
	Bug Reports. Anything about the product that threatens its value.
	Issues. Anything about the project that threatens its value.
	Test Coverage Outline. Aspects of the product we might want to test.
	Risks. Any potential areas of bugginess or types of bug.
	Test Data. Any data developed for use in tests.
	Test Tools. Any tools acquired or developed to aid testing.
	Test Strategy. The set of ideas that guide our test design.
	Test Infrastructure and Lab Procedures. General practices, protocols, controls, and systems that provide a basis for excellent testing.
	Test Estimation. Ideas about what we need and how much time we need.
	Testing Story. What we know about our testing, so far.
	Product Story. What we know about the product, so far.
	Test Process Assessment. Our own assessment of the quality of our test process.
	Tester. The tester evolves over the course of the project.
	Test Team. The test team gets better, too.
	Developer and Customer Relationships. As you test, you also get to know the people you are working with.

Exploration Skills

These are the skills that comprise professional and cost effective exploration of technology. Each is distinctly observable and learnable, and each is necessary for excellent exploratory work:

Self-Management

	Chartering your work. Making decisions about what you will work on and how you will work. Understanding your client's needs, the problems you must solve, and assuring that your work is on target.
	Establishing procedures and protocols. Designing ways of working that allow you to manage your study productively. This also means becoming aware of critical patterns, habits, and behaviors that may be intuitive and bringing them under control.
	Establishing the conditions you need to succeed. Wherever feasible and to the extent feasible, establish control over the surrounding environment such that your tests and observations will not be disturbed by extraneous and uncontrolled factors.
	Maintaining self-awareness. Monitoring your emotional, physical, and mental states as they influence your exploration.
	Behaving ethically. Understanding and fulfilling your responsibilities under any applicable ethical code during the course of your exploration.
	Monitoring issues in the exploration. Maintaining an awareness of potential problems, obstacles, limitations and biases in your exploration. Understanding the cost vs. value of the work.
	Branching your work and backtracking. Allowing yourself to be productively distracted from a course of action to explore an unanticipated new idea. Identifying opportunities and pursuing them without losing track of your process.
	Focusing your work. Isolating and controlling factors to be studied. Repeating experiments. Limiting change. Precise observation. Defining and documenting procedures. Using focusing heuristics.
	De-focusing your work. Including more factors in your study. Diversifying your work. Changing many factors at once. Broad observation. Trying new procedures. Using defocusing heuristics.
	Alternating activities to improve productivity. Switching among different activities or perspectives to create or relieve productive tension and make faster progress. See *Exploratory Testing Polarities*.
	Maintaining useful and concise records. Preserving information about your process, progress, and findings. Note-taking.
	Deciding when to stop. Selecting and applying stopping heuristics to determine when you have achieved good enough progress and results, or when your exploration is no longer worthwhile.

Collaboration

	Getting to know people. Meeting and learning about the people around you who might be helpful, or whom you might help. Developing a collegial network within your project and beyond it.
	Conversation. Talking through and elaborating ideas with other people.
	Serving other testers. Performing services that support the explorations of other testers on their own terms.
	Guiding other testers. Supervising testers who support your explorations. Coaching testers.
	Asking for help. Articulating your needs and negotiating for assistance.
	Telling the story of your exploration. Making a credible, professional report of your work to your clients in oral and written form that explains and justifies what you did.
	Telling the product story. Making a credible, relevant account of the status of the object you are studying, including bugs found. This is the ultimate goal for most test projects.

Learning

	Discovering and developing resources. Obtaining information or facilities to support your effort. Exploring those resources.
	Applying technical knowledge. Surveying what you know about the situation and technology and applying that to your work. An expert in a specific kind of technology or application may explore it differently.
	Considering history. Reviewing what's been done before and mining that resource for better ideas.
	Using Google and the Web. Of course, there are many ways to perform research on the Internet. But, acquiring the technical information you need often begins with Google.
	Reading and analyzing documents. Reading carefully and analyzing the logic and ideas within documents that pertain to your subject.
	Asking useful questions. Identifying missing information, conceiving of questions, and asking questions in a way that elicits the information you seek.
	Pursuing an inquiry. A line of inquiry is a structure that organizes reading, questioning, conversation, testing, or any other information gathering tactic. It is investigation oriented around a *specific* goal. Many lines of inquiry may be served during exploration. This is, in a sense, the opposite of practicing curiosity.
	Indulging curiosity. Curiosity is investigation oriented around this *general* goal: to learn something that might be useful, at some later time. This is, in a sense, the opposite of pursuing a line of inquiry.
	Generating and elaborating a requisite variety of ideas. Working quickly in a manner good enough for the circumstances. Revisiting the solution later to extend, refine, refactor or correct it.
	Overproducing ideas for better selection. Producing many different speculative ideas and making speculative experiments, more than you can elaborate upon in the time you have. Examples are brainstorming, trial and error, genetic algorithms, free market dynamics.
	Abandoning ideas for faster progress. Letting go of some ideas in order to focus and make progress with other ones.
	Recovering or reusing ideas for better economy. Revisiting your old ideas, models, questions or conjectures; or discovering them already made by someone else.

Testing

	Applying tools. Enabling new kinds of work or improving existing work by developing and deploying tools.
	Interacting with your subject. Making and managing contact with the subject of your study; for technology, configuring and operating it so that it demonstrates what it can do.
	Creating models and identifying relevant factors for study. Composing, decomposing, describing, and working with mental models of the things you are exploring. Identifying relevant dimensions, variables, and dynamics.
	Discovering and characterizing elements and relationships within the product. Analyze consistencies, inconsistencies, and any other patterns within the subject of your study.
	Conceiving and describing your conjectures. Considering possibilities and probabilities. Considering multiple, incompatible explanations that account for the same facts. Inference to the best explanation.
	Constructing experiments to refute your conjectures. As you develop ideas about what's going on, creating and performing tests designed to disconfirm those beliefs, rather than repeating the tests that merely confirm them.
	Making comparisons. Studying things in the world with the goal of identifying and evaluating relevant differences and similarities between them.
	Detecting potential problems. Designing and applying oracles to detect behaviors and attributes that may be trouble.

	Observing what is there. Gathering empirical data about the object of your study; collecting different kinds of data, or data about different aspects of the object; establishing procedures for rigorous observations.
	Noticing what is missing. Combining your observations with your models to notice the significant absence of an object, attribute, or pattern.

Exploratory Testing Polarities

To develop ideas or search a complex space quickly yet thoroughly, not only must you look at the world from many points of view and perform many kinds of activities (which may be polar opposites), but your mind may get sharper from the very act of switching from one kind of activity to another. Here is a partial list of polarities:

	Warming up vs. cruising vs. cooling down
	Doing vs. describing
	Doing vs. thinking
	Careful vs. quick
	Data gathering vs. data analysis
	Working with the product vs. reading about the product
	Working with the product vs. working with the developer
	Training (or learning) vs. performing
	Product focus vs. project focus
	Solo work vs. team effort
	Your ideas vs. other peoples' ideas
	Lab conditions vs. field conditions
	Current version vs. old versions
	Feature vs. feature
	Requirement vs. requirement
	Coverage vs. oracles
	Testing vs. touring
	Individual tests vs. general lab procedures and infrastructure
	Testing vs. resting
	Playful vs. serious

Test Strategy

This is a compressed version of the Satisfice Heuristic Test Strategy model. It's a set of considerations designed to help you test robustly or evaluate someone else's testing.

Project Environment

- ❑ *Mission.* The problems you are commissioned to solve for your customer.
- ❑ *Information.* Information about the product or project that is needed for testing.
- ❑ *Developer Relations.* How you get along with the programmers.
- ❑ *Test Team.* Anyone who will perform or support testing.
- ❑ *Equipment & Tools.* Hardware, software, or documents required to administer testing.
- ❑ *Schedules.* The sequence, duration, and synchronization of project events.
- ❑ *Test Items.* The product to be tested.
- ❑ *Deliverables.* The observable products of the test project.

Product Elements

- ❑ *Structure.* Everything that comprises the physical product.
- ❑ *Functions.* Everything that the product does.
- ❑ *Data.* Everything that the product processes.
- ❑ *Platform.* Everything on which the product depends (and that is outside your project).
- ❑ *Operations.* How the product will be used.
- ❑ *Time.* Any relationship between the product and time.

Quality Criteria Categories

- ❑ *Capability.* Can it perform the required functions?
- ❑ *Reliability.* Will it work well and resist failure in all required situations?
- ❑ *Usability.* How easy is it for a real user to use the product?
- ❑ *Security.* How well is the product protected against unauthorized use or intrusion?
- ❑ *Scalability.* How well does the deployment of the product scale up or down?
- ❑ *Performance.* How speedy and responsive is it?
- ❑ *Installability.* How easily can it be installed onto it target platform?
- ❑ *Compatibility.* How well does it work with external components & configurations?
- ❑ *Supportability.* How economical will it be to provide support to users of the product?
- ❑ *Testability.* How effectively can the product be tested?
- ❑ *Maintainability.* How economical is it to build, fix or enhance the product?
- ❑ *Portability.* How economical will it be to port or reuse the technology elsewhere?
- ❑ *Localizability.* How economical will it be to publish the product in another language?

General Test Techniques

- ❑ *Function Testing.* Test what it can do.
- ❑ *Domain Testing.* Divide and conquer the data.
- ❑ *Stress Testing.* Overwhelm the product.
- ❑ *Flow Testing.* Do one thing after another.
- ❑ *Scenario Testing.* Test to a compelling story.
- ❑ *Claims Testing.* Verify every claim.
- ❑ *User Testing.* Involve the users.
- ❑ *Risk Testing.* Imagine a problem, then find it.
- ❑ *Automatic Checking.* Write a program to generate and run a zillion checks.

Test Framing

Michael Bolton
http://www.developsense.com
SQDG
February 2011

Acknowledgements

- Ideas about test framing have been developed in collaboration with James Bach
- Some material in this presentation is from Rapid Software Testing, a course by James Bach and Michael Bolton.
  - http://www.satisfice.com/rst.pdf
  - http://www.satisfice.com/rst-appendices.pdf

Important Questions

Why run that test?

- Variations:
  - Why are you planning to run that test?
  - Why are you running that test *right now*?
  - Why did you run that test?
  - Why perform that step?
  - Why make that observation?

Important Questions

Why NOT run THAT test?

- Variations:
  - Why aren't you planning to run that test?
  - Why aren't you running that test *right now*?
  - Why didn't you run that test?
  - Why not perform this step?
  - Why not make that observation?

Important Questions

Why didn't you find that bug?

- Variations:
  - Why didn't you find that bug earlier?
  - Why did you apparently ignore that requirement?
  - Why did you miss that symptom?
  - Why did you misinterpret that symptom?

Important Questions

Why do you think that's a bug?

- Variations:
  - Why do you say that this isn't working properly?
  - What requirement is being left unfulfilled here?
  - Why do you think that's a requirement?
  - For whom might this be a problem?
  - Do you think a user would ever do that?

Even more generally…

Why are you doing this?

- Variations:
  - Why are you not doing that?
  - How does this test relate to a requirement?
  - How does this test relate to a risk?
  - How does this test relate to your mission?

What is testing?

Getting Answers

"Try it and see if it works."

Procedures	Coverage	Oracles
Get different versions	Where to look?	Read the spec
Set them up	How to look?	(There's no spec? Oh.)
Try simple things	What's there?	(There IS a spec! Oh, it's old and wrong.)
Try complex things	What's not there?	Find inconsistencies
Try sequences	What's invisible?	Find "obvious" problems
Try combinations	Did it change?	Find obscure problems
Try weird things	Will it change?	Find BAD problems
Try them again	How 'bout now?	

8

What is testing?

Getting answers…

"Try it and see if it works."

really means…

"**Try it to learn,**
sufficiently, everything that matters
about whether it can work and
how it might not work."

9

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.

"everything that matters"

Know your mission.

10

What is Testing?

"Try it to learn…how it might not work."

1. Model the test space.
2. Determine coverage.
3. Determine oracles.
4. Determine test procedures.
5. Configure the test system.
6. Operate the test system.
7. Observe the test system.
8. Evaluate the test results.
9. Apply a stopping heuristic.
10. Report test results.

Risk!

Most testing is driven by questions about risk…

…So, it helps to relate test results back to risk.

11

To test is to compose, edit, narrate, and justify *three* parallel stories.

1. You must tell a story about the product…
   - …about how it failed, and how it *might* fail...
   - …in ways that matter to your various clients.
2. But also tell a story about how you tested it…
   - …how you configured, operated and observed it…
   - …about what you haven't tested yet…
   - …or won't test at all…
3. And also tell a story that explains how good your testing was…
   - …why your testing has been good enough…
   - …why what you haven't done (so far) doesn't matter…
   - …what the risks and costs of testing are…
   - …how testable (or not) the product is…
   - …what you need and what you recommend.

12

What is test framing?

Test framing is *the chain of logical connections that structure and inform a test,* from the mission to the test result

Given this…

Mission

If *these premises and facts* are true…

…and if we observe *this specific behaviour…*

…and we apply *this relevant oracle…*

…then we can make *these conclusions* about the...

Test Result

Framing ~= Traceability

- Framing is, in essence, traceability…
- …but typically we hear people talk of traceability in an impoverished way: between *tests* and requirements *documents*
- *Can you demonstrate traceability between tests and implicit requirements?*

Much More Traceability

1. Product traces to specifications.
2. Specifications trace to standards.
3. Test sessions trace to product versions.
4. Test sessions trace to specifications.
5. Test sessions trace to logs which trace to product, playbook and specifications.
6. Tests trace to claims.
7. Test sessions trace to charters and charters to playbook.
7. Playbook traces to standards.
8. Playbook traces to specifications.
9. Playbook traces to risks which trace to specifications…
10. Tests trace to risk…
11. Tests trace to implicit requirements…
12. Tests trace to other tests…

Vocabulary

- system
  - a set of things in meaningful relationship
- structure
  - that which forms the unchanging parts and relationships of a system; a consistent pattern for the system; “that which remains”
- narration
  - telling a story that fits in time
- framing
  - via logic and narrative, placing the test in logical relationship with the structures that inform it

Vocabulary

- logic
  - a formal means of convincing or proving facts via valid arguments
  - …using a set of propositions linked by connectives or operators
- proposition
  - a simple statement of fact or inference
- connectives
  - formal logic: if, and, or, and not, else, if and only if, therefore
  - less formally: because, unless, otherwise…

Framing provides traceability, but testers often limit traceability as being between *tests* and requirements *documents*—explicit requirements.

Can you demonstrate traceability between tests and implicit requirements?

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.”

18

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. Boot the test system."
- "Step 2. Start the application."

19

Testing against requirements is all about modeling.

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."

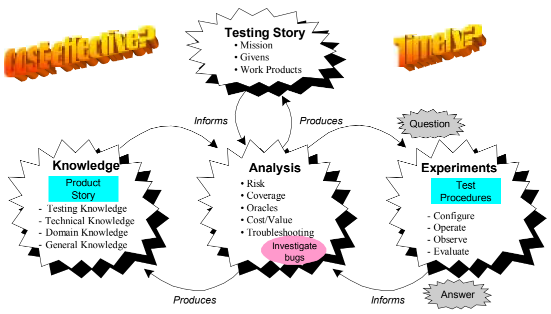
How Do We Know What "Is"?

We see the signs!

"If I see X, then *probably* Y, because *probably* A, B, C, D, etc."

- THIS CAN FAIL:
  - Getting into a car
    - Ooops, not my car.
  - Drunk driving
    - People with diabetes can seem drunk
  - Irrational decisions
    - rational from a different set of values
  - I can never find the sugar
    - I have a preset model for sugar—and it's always the other one
  - Hotel offers two clean beds
    - That's not exactly what they said

One View of Testing Structure



22

Project Environment
Aspects of Our Context

"CIDTESTD – Mother Approved!"

- Customers
  - *Anyone who is a client of the test project.*
- 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.*

23

Quality Criteria
Identifying Value and Threats To It

CRUSSPIC STMPL

- Capability
- Reliability
- Usability
- Security
- Scalability
- Performance
- Installability
- Compatibility
- Supportability
- Testability
- Maintainability
- Portability
- Localizability

Many test approaches focus on Capability (functionality) and underemphasize the other criteria

24

Product Elements
Ways to Model and Cover The Product

"SFDPOT - San Francisco DePOT"

- Structure
  - *What are the pieces and how do they fit together?*
- Function
  - *What does the product do?*
- Data
  - *What does the product do things to?*
- Platform
  - *What does the product depend upon?*
- Operations
  - *How do people actually use the program?*
- Time
  - *How does the product interact with time?*

25

Test Techniques
General Ways to Test

FDSFSCURA

- Function testing
  - *Test what it does*
- Domain testing
  - *Divide and conquer the data*
- Stress testing
  - *Overwhelm or starve the product*
- Flow testing
  - *Do one thing after another after another*
- Scenario testing
  - *Test to a compelling story*

26

Test Techniques
General Ways to Test

FDSFSCURA

- Claims testing
  - *Test everything that people say it should do*
- User testing
  - *Involve the users (or systematically simulate them)*
- Risk testing
  - *Imagine a problem, and then look for it*
- Automatic testing
  - *Perform zillions of tests, aided by machines*

27

Thirty-Six Test Strategy Heuristics

"cidtestdsfdpotcrusspicstmplfdsfscura"

Project Environment: Customers, Information, Developer relations, Team, Equipment & tools, Schedule, Test Items, Deliverables

Product Elements: Structures, Functions, Data, Platforms, Operations, Time

Quality Criteria: Capability, Reliability, Usability, Security, Scalability, Performance, Installability, Compatibility, Supportability, Testability, Maintainability, Portability, Localizability

Test Techniques: Function testing, Domain testing, Stress testing, Flow testing, Scenario testing, Claims testing, User testing, Risk testing, Automatic testing

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

- **History:** The present version of the system *is consistent* with past versions of it.
- **Image:** The system *is consistent* with an image that the organization wants to project.
- **Comparable Products:** The system *is consistent* with comparable systems.
- **Claims:** The system *is consistent* with what important people say it's supposed to be.
- **Users' Expectations:** The system *is consistent* with what users want.
- **Product:** Each element of the system is *consistent* with comparable elements in the same system.
- **Purpose:** The system *is consistent* with its purposes, both explicit and implicit.
- **Statutes:** The system *is consistent* with applicable laws.
- **Familiarity:** The system ***is not consistent*** with the pattern of any familiar problem.

Consistency heuristics rely on the quality of your models of the product and its context.

29

What if you have an unframed test?

Try framing it!

But if you can't do it perfectly, or right away, that might be okay. Why?

To test a *very simple* product meticulously,
part of a complex product meticulously,
or to maximize test *integrity*…

Focus!

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).

How can you justify an unframed test?

All of the hidden frames!

Focus in testing is valuable, but we must also practice de-focusing to expose new problems and to trigger new ideas about risk.

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

De-Focus!

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 *hard to pass*, instead of easy to reproduce.

Galumphing

A Defocusing Heuristic to Exploit Variability

- doing things in a deliberately over-elaborate way
- adding lots of unnecessary but inert actions that are inexpensive and shouldn't (in theory) affect the test outcome
  - bring up a dialog and dismiss it
  - modify an option and rescind it
  - perform an action and reverse it
  - re-selecting default options
  - inserting an expression where a single value would do
  - over-filling an input field, then fixing it

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 (but be careful; humans almost always act non-randomly)
- **Data Substitution**
  - The same actions may have dramatically different results when tried on a different database, or with different input.
- **Timing/Concurrency Variations**
  - The same actions may have different results depending on the time frame in which they occur and other concurrent events.
- **Platform Substitution**
  - Supposedly equivalent platforms may not be.

Exploiting Variation To Find More Bugs

- **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.