Test Framing

A Brief Introduction

Michael Bolton
EuroSTAR 2010
Copenhagen, Denmark

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

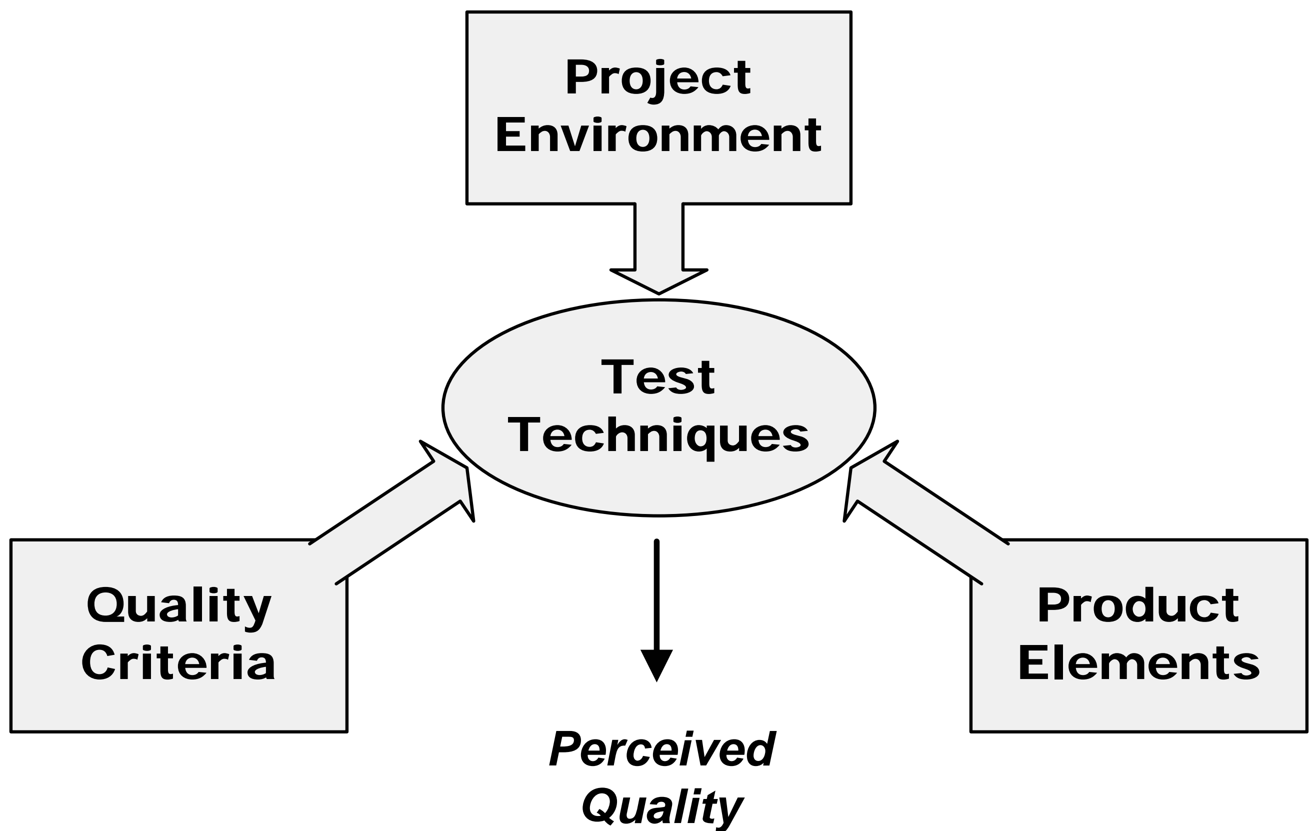
Test Techniques: See the "Test Techniques" portion of the Heuristic Test Strategy Model. Also see Lee Copeland, A Practitioner's Guide to Software Test Design, Artech House Publishers, 2003.

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

Designed by James Bach
james@satisfice.com
www.satisfice.com
Copyright 1996-2006, Satisfice, Inc.

Version 4.8
3/28/2006

Heuristic Test Strategy Model



The **Heuristic Test Strategy Model** is a set of patterns for designing a test strategy. The immediate purpose of this model is to remind testers of what to think about when they are creating tests. Ultimately, it is intended to be customized and used to facilitate dialog, self-directed learning, and more fully conscious testing among professional testers.

Project Environment includes resources, constraints, and other forces in the project that enable us to test, while also keeping us from doing a perfect job. Make sure that you make use of the resources you have available, while respecting your constraints.

Product Elements are things that you intend to test. Software is so complex and invisible that you should take special care to assure that you indeed examine all of the product that you need to examine.

Quality Criteria are the rules, values, and sources that allow you as a tester to determine if the product has problems. Quality criteria are multidimensional, and often hidden or self-contradictory.

Test Techniques are strategies for creating tests. All techniques involve some sort of analysis of project environment, product elements, and quality criteria.

Perceived Quality is the result of testing. You can never know the "actual" quality of a software product, but through the application of a variety of tests, you can derive an informed assessment of it.

General Test Techniques

A test technique is a way of creating tests. There are many interesting techniques. The list includes nine general *techniques. By "general technique" I mean that the technique is simple and universal enough to apply to a wide variety of contexts. Many specific techniques are based on one or more of these nine. And an endless variety of specific test techniques can be constructed by combining one or more general techniques with coverage ideas from the other lists in the Heuristic Test Strategy Model.*

Function Testing

Test what it can do

1. Identify things that the product can do (functions and sub-functions).
2. Determine how you'd know if a function was capable of working.
3. Test each function, one at a time.
4. See that each function does what it's supposed to do and not what it isn't supposed to do.

Domain Testing

Divide and conquer the data

1. Look for any data processed by the product. Look at outputs as well as inputs.
2. Decide which particular data to test with. *Consider things like boundary values, typical values, convenient values, invalid values, or best representatives.*
3. Consider combinations of data worth testing together.

Stress Testing

Overwhelm the product

1. Look for sub-systems and functions that are vulnerable to being overloaded or "broken" in the presence of challenging data or constrained resources.
2. Identify data and resources related to those sub-systems and functions.
3. Select or generate challenging data, or resource constraint conditions to test with: *e.g., large or complex data structures, high loads, long test runs, many test cases, low memory conditions.*

Flow Testing

Do one thing after another

1. Define test procedures or high level cases that incorporate multiple activities connected end-to-end.
2. Don't reset the system between tests.
3. Vary timing and sequencing, and try parallel threads.

Scenario Testing

Test to a compelling story

1. Begin by thinking about everything going on *around* the product.
2. Design tests that involve meaningful and complex interactions with the product.
3. A good scenario test is a compelling story of how someone who matters might do something that matters with the product.

Claims Testing

Verify every claim

1. Identify reference materials that include claims about the product (implicit or explicit).
2. Analyze individual claims, and clarify vague claims.
3. Verify that each claim about the product is true.
4. If you're testing from an explicit specification, expect it and the product to be brought into alignment.

User Testing

Involve the users

1. Identify categories and roles of users.
2. Determine what each category of user will do (use cases), how they will do it, and what they value.
3. Get real user data, or bring real users in to test.
4. Otherwise, systematically simulate a user (be careful—it's easy to think you're like a user even when you're not).
5. Powerful user testing is that which involves a variety of users and user roles, not just one.

Risk Testing

Imagine a problem, then look for it.

1. What kinds of problems could the product have?
2. Which kinds matter most? Focus on those.
3. How would you detect them if they were there?
4. Make a list of interesting problems and design tests specifically to reveal them.
5. It may help to consult experts, design documentation, past bug reports, or apply risk heuristics.

Automatic Testing

Run a million different tests

1. Look for opportunities to automatically generate a lot of tests.
2. Develop an automated, high speed evaluation mechanism.
3. Write a program to generate, execute, and evaluate the tests.

Project Environment

Creating and executing tests is the heart of the test project. However, there are many factors in the project environment that are critical to your decision about what particular tests to create. In each category, below, consider how that factor may help or hinder your test design process. Try to exploit every resource.

- ❑ **Customers.** *Anyone who is a client of the test project.*
  - Do you know who your customers are? Whose opinions matter? Who benefits or suffers from the work you do?
  - Do you have contact and communication with your customers? Maybe they can help you test.
  - Maybe your customers have strong ideas about what tests you should create and run.
  - Maybe they have conflicting expectations. You may have to help identify and resolve those.
- ❑ **Information.** *Information about the product or project that is needed for testing.*
  - Are there any engineering documents available? User manuals? Web-based materials?
  - Does this product have a history? Old problems that were fixed or deferred? Pattern of customer complaints?
  - Do you need to familiarize yourself with the product more, before you will know how to test it?
  - Is your information current? How are you apprised of new or changing information?
  - Is there any complex or challenging part of the product about which there seems strangely little information?
- ❑ **Developer Relations.** *How you get along with the programmers.*
  - *Hubris:* Does the development team seem overconfident about any aspect of the product?
  - *Defensiveness:* Is there any part of the product the developers seem strangely opposed to having tested?
  - *Rapport:* Have you developed a friendly working relationship with the programmers?
  - *Feedback loop:* Can you communicate quickly, on demand, with the programmers?
  - *Feedback:* What do the developers think of your test strategy?
- ❑ **Test Team.** *Anyone who will perform or support testing.*
  - Do you know who will be testing?
  - Are there people not on the “test team” that might be able to help? People who’ve tested similar products before and might have advice? Writers? Users? Programmers?
  - Do you have enough people with the right skills to fulfill a reasonable test strategy?
  - Are there particular test techniques that the team has special skill or motivation to perform?
  - Is any training needed? Is any available?
- ❑ **Equipment & Tools.** *Hardware, software, or documents required to administer testing.*
  - *Hardware:* Do we have all the equipment you need to execute the tests? Is it set up and ready to go?
  - *Automation:* Are any test automation tools needed? Are they available?
  - *Probes:* Are any tools needed to aid in the observation of the product under test?
  - *Matrices & Checklists:* Are any documents needed to track or record the progress of testing?
- ❑ **Schedule.** *The sequence, duration, and synchronization of project events.*
  - *Test Design:* How much time do you have? Are there tests better to create later than sooner?
  - *Test Execution:* When will tests be executed? Are some tests executed repeatedly, say, for regression purposes?
  - *Development:* When will builds be available for testing, features added, code frozen, etc.?
  - *Documentation:* When will the user documentation be available for review?
- ❑ **Test Items.** *The product to be tested.*
  - *Scope:* What parts of the product are and are not within the scope of your testing responsibility?
  - *Availability:* Do you have the product to test?
  - *Volatility:* Is the product constantly changing? What will be the need for retesting?
  - *New Stuff:* What has recently been changed or added in the product?
  - *Testability:* Is the product functional and reliable enough that you can effectively test it?
  - *Future Releases:* What part of your tests, if any, must be designed to apply to future releases of the product?
- ❑ **Deliverables.** *The observable products of the test project.*
  - *Content:* What sort of reports will you have to make? Will you share your working notes, or just the end results?
  - *Purpose:* Are your deliverables provided as part of the product? Does anyone else have to run your tests?
  - *Standards:* Is there a particular test documentation standard you’re supposed to follow?
  - *Media:* How will you record and communicate your reports?

Product Elements

Ultimately a product is an experience or solution provided to a customer. Products have many dimensions. So, to test well, we must examine those dimensions. Each category, listed below, represents an important and unique aspect of a product. Testers who focus on only a few of these are likely to miss important bugs.

❑ **Structure.** *Everything that comprises the physical product.*

- *Code:* the code structures that comprise the product, from executables to individual routines.
- *Interfaces:* points of connection and communication between sub-systems.
- *Hardware:* any hardware component that is integral to the product.
- *Non-executable files:* any files other than multimedia or programs, like text files, sample data, or help files.
- *Collateral*: anything beyond software and hardware that is also part of the product, such as paper documents, web links and content, packaging, license agreements, etc..

❑ **Functions.** *Everything that the product does.*

- *User Interface:* any functions that mediate the exchange of data with the user (e.g. navigation, display, data entry).
- *System Interface:* any functions that exchange data with something other than the user, such as with other programs, hard disk, network, printer, etc.
- *Application:* any function that defines or distinguishes the product or fulfills core requirements.
- *Calculation:* any arithmetic function or arithmetic operations embedded in other functions.
- *Time-related:* time-out settings; daily or month-end reports; nightly batch jobs; time zones; business holidays; interest calculations; terms and warranty periods; chronograph functions.
- *Transformations:* functions that modify or transform something (e.g. setting fonts, inserting clip art, withdrawing money from account).
- *Startup/Shutdown:* each method and interface for invocation and initialization as well as exiting the product.
- *Multimedia:* sounds, bitmaps, videos, or any graphical display embedded in the product.
- *Error Handling:* any functions that detect and recover from errors, including all error messages.
- *Interactions:* any interactions or interfaces between functions within the product.
- *Testability:* any functions provided to help test the product, such as diagnostics, log files, asserts, test menus, etc.

❑ **Data.** *Everything that the product processes.*

- *Input:* any data that is processed by the product.
- *Output:* any data that results from processing by the product.
- *Preset:* any data that is supplied as part of the product, or otherwise built into it, such as prefabricated databases, default values, etc.
- *Persistent*: any data that is stored internally and expected to persist over multiple operations. This includes modes or states of the product, such as options settings, view modes, contents of documents, etc.
- *Sequences:* any ordering or permutation of data, e.g. word order, sorted vs. unsorted data, order of tests.
- *Big and little:* variations in the size and aggregation of data.
- *Noise:* any data or state that is invalid, corrupted, or produced in an uncontrolled or incorrect fashion.
- *Lifecycle:* transformations over the lifetime of a data entity as it is created, accessed, modified, and deleted.

❑ **Platform.** *Everything on which the product depends (and that is outside your project).*

- *External Hardware:* hardware components and configurations that are not part of the shipping product, but are required (or optional) in order for the product to work: CPU's, memory, keyboards, peripheral boards, etc.
- *External Software:* software components and configurations that are not a part of the shipping product, but are required (or optional) in order for the product to work: operating systems, concurrently executing applications, drivers, fonts, etc.
- *Internal Components:* libraries and other components that are embedded in your product but are produced outside your project. Since you don't control them, you must determine what to do in case they fail.

❑ **Operations.** *How the product will be used.*

- *Users:* the attributes of the various kinds of users.
- *Environment:* the physical environment in which the product operates, including such elements as noise, light, and distractions.
- *Common Use:* patterns and sequences of input that the product will typically encounter. This varies by user.
- *Disfavored Use:* patterns of input produced by ignorant, mistaken, careless or malicious use.
- *Extreme Use:* challenging patterns and sequences of input that are consistent with the intended use of the product.

❑ **Time.** *Any relationship between the product and time.*

- *Input/Output:* when input is provided, when output created, and any timing relationships (delays, intervals, etc.) among them.
- *Fast/Slow:* testing with "fast" or "slow" input; fastest and slowest; combinations of fast and slow.
- *Changing Rates:* speeding up and slowing down (spikes, bursts, hangs, bottlenecks, interruptions).
- *Concurrency*: more than one thing happening at once (multi-user, time-sharing, threads, and semaphores, shared data).

Quality Criteria Categories

A quality criterion is some requirement that defines what the product should be. By looking thinking about different kinds of criteria, you will be better able to plan tests that discover important problems fast. Each of the items on this list can be thought of as a potential risk area. For each item below, determine if it is important to your project, then think how you would recognize if the product worked well or poorly in that regard.

Operational Criteria

- ❑ **Capability.** *Can it perform the required functions?*
- ❑ **Reliability.** *Will it work well and resist failure in all required situations?*
  - *Error handling:* the product resists failure in the case of errors, is graceful when it fails, and recovers readily.
  - *Data Integrity:* the data in the system is protected from loss or corruption.
  - *Safety:* the product will not fail in such a way as to harm life or property.
- ❑ **Usability.** *How easy is it for a real user to use the product?*
  - *Learnability:* the operation of the product can be rapidly mastered by the intended user.
  - *Operability:* the product can be operated with minimum effort and fuss.
  - *Accessibility:* the product meets relevant accessibility standards and works with O/S accessibility features.
- ❑ **Security.** *How well is the product protected against unauthorized use or intrusion?*
  - *Authentication*: the ways in which the system verifies that a user is who she says she is.
  - *Authorization*: the rights that are granted to authenticated users at varying privilege levels.
  - *Privacy*: the ways in which customer or employee data is protected from unauthorized people.
  - *Security holes*: the ways in which the system cannot enforce security (e.g. social engineering vulnerabilities)
- ❑ **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 its target platform(s)?*
  - *System requirements:* Does the product recognize if some necessary component is missing or insufficient?
  - *Configuration:* What parts of the system are affected by installation? Where are files and resources stored?
  - *Uninstallation:* When the product is uninstalled, is it removed cleanly?
  - *Upgrades:* Can new modules or versions be added easily? Do they respect the existing configuration?
- ❑ **Compatibility.** *How well does it work with external components & configurations?*
  - *Application Compatibility:* the product works in conjunction with other software products.
  - *Operating System Compatibility*: the product works with a particular operating system.
  - *Hardware Compatibility*: the product works with particular hardware components and configurations.
  - *Backward Compatibility:* the products works with earlier versions of itself.
  - *Resource Usage:* the product doesn't unnecessarily hog memory, storage, or other system resources.

Development Criteria

- ❑ **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?*
  - *Regulations*: Are there different regulatory or reporting requirements over state or national borders?
  - *Language:* Can the product adapt easily to longer messages, right-to-left, or ideogrammatic script?
  - *Money:* Must the product be able to support multiple currencies? Currency exchange?
  - *Social or cultural differences:* Might the customer find cultural references confusing or insulting?

Heuristic Test Planning: Context Model

Designed by James Bach, http://www.satisfice.com
Copyright (c) 2000, Satisfice, Inc.

v1.2

Missions

- Find Important Problems
- Assess Quality/Risk
- Certify to Standard
- Fulfill Process Mandates
- Satisfy Stakeholders
- Assure Accountability
- Advise about QA
- Advise about Testing
- Advise about Quality
- Maximize Efficiency
- Minimize Cost
- Minimize Time

Development

- Product
- Project Lifecycle
- Project Management
- Configuration Management
- Defect Prevention
- Development Team

Requirements

- Product Mission
- Stakeholders
- Quality Criteria
- Reference Material

Test Process

- Strategy
- Logistics
- Products

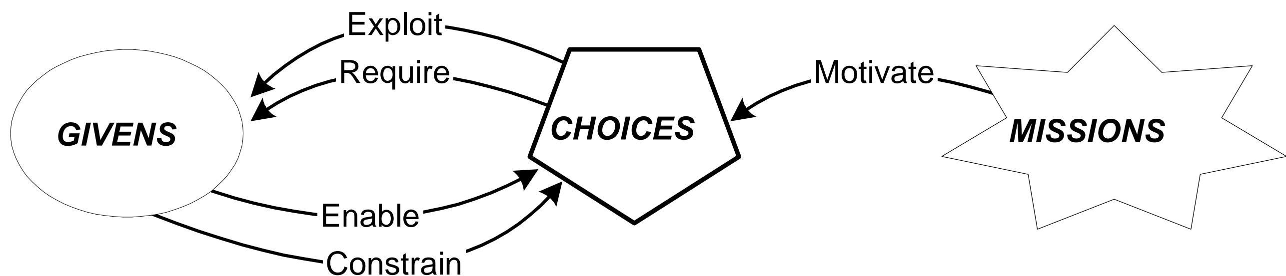
Test Team

- Expertise
- Loading
- Cohesion
- Motivation
- Leadership
- Project Integration

Test Lab

- Test Platforms
- Test Tools
- Test Library
- Problem Tracking System
- Office Facilities

How Context Influences the Test Plan



Context-Driven Planning

1. Understand who is involved in the project and how they matter.

2. Understand and negotiate the GIVENS so that you understand the constraints on your work, understand the resources available, and can test effectively.

3. Negotiate and understand the MISSIONS of testing in your project.

4. Make CHOICES about how to test that exploit the GIVENS and allow you to achieve your MISSIONS.

5. Monitor the status of the project and continue to adjust the plan as needed to maintain congruence among GIVENS, CHOICES, and MISSIONS.

Test Process Choices

We testers and test managers don't often have a lot of control over the context of our work. Sometimes that's a problem. A bigger problem would be not having control over the work itself. When a test process is controlled from outside the test team, it's likely to be much less efficient and effective. This model is designed with the assumption that there are three elements over which you probably have substantial control: *test strategy, test logistics,* and *test products*. Test planning is mainly concerned with designing these elements of test process to work well within the context.

Test strategy is how you cover the product and detect problems. You can't test everything in every way, so here's where you usually have the most difficult choices.

Test logistics is how and when you apply resources to execute the test strategy. This includes how you coordinate with other people on the project, who is assigned to what tasks, etc.

Test products are the materials and results you produce that are visible to the clients of testing. These may include test scripts, bug reports, test reports, or test data to name a few.

Exploratory Testing Dynamics

Created by James Bach, Jonathan Bach, and Michael Bolton[1] *v2.2* *Copyright © 2005-2009, Satisfice, Inc.*

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 following are some of the many dynamics that comprise exploratory testing.

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.

[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

Exploration Skills and Tactics

These are the skills that comprise professional and cost effective exploration of technology. Each is distinctly observable and learnable, and each is necessary to 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.
	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.
	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.

Developing Ideas

	Discovering and developing resources. Obtaining information or facilities to support your effort. Exploring those resources and developing relationships with people and organizations that can help you.
	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.
	Collaborating for better ideas. Working and thinking with other people on the same problem; elaborating other people's 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 a line of 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.
	Practicing 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. Revisiting your old ideas, models, questions or conjectures; or discovering them already made by someone else.

Examining a Product

	Applying tools. Enabling new kinds of work or improving existing work by developing and deploying tools.
	Interacting with the product. Making and managing contact with the object of your study; 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 objects 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.
	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

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