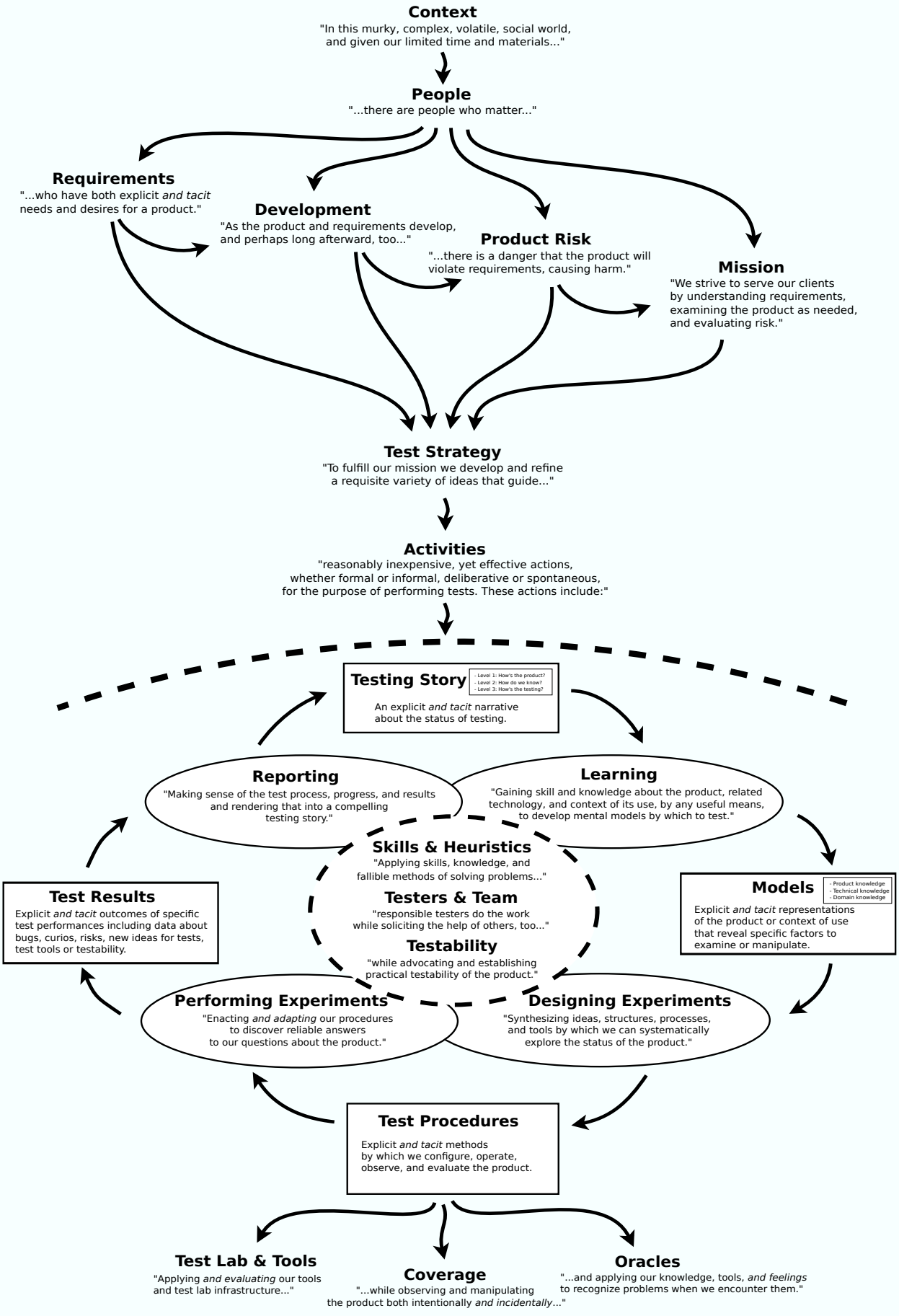
Appendices, v5.5

James Bach
james@satisfice.com
www.satisfice.com

Michael Bolton
michael@developsense.com
www.developsense.com

RST Appendices

A Rapid Testing Framework



How is Rapid Testing different from "Factory Style" testing?

	Factory Style (e.g. ISTQB, TMap, TPI, ISO/IEEE standards, Six Sigma, TQM, CMM, RUP)	Rapid Software Testing
Basic Idea	Follow industry consensus ("best practices") and principles of efficient manufacturing. Products are technological artifacts. Testing is a repetitive process of collecting facts about products. Humans are relatively unreliable, however, and skilled people are expensive and hard to find. It's therefore important to use a testing methodology that minimizes reliance on subjective factors and tester creativity. We can do this by structuring tasks in a manner reminiscent of a manufacturing plant—explicitly defining procedures and then monitoring adherence to those procedures.	Fulfill our mission for our clients by developing and applying skills and heuristics. Products are solutions that fulfill some need. Testing is an adaptive process of learning and analysis involving a great variety of experiments, observations and inferences about products. Although it is not possible to fully define or formalize that process, skilled humans are uniquely able to perform it. We therefore choose to use a methodology that maximizes freedom, self-regulation, and responsibility. We can do this by structuring training and culture in a manner reminiscent of hospitals, law firms, or elite military units—using practical drills, realistic missions, and other scalable training methods (along with expert mentoring) to build skills and create a culture of excellence. The people who do the work then structure their processes as needed.
Notion of Product Quality	A product is a system of elements and behaviors that fulfill explicitly defined requirements. The quality of a product is how well it conforms to those requirements. Quality should be measured objectively.	A product is a system of elements and behaviors, created by people, that creates a desirable experience or solution for other people. A product is always produced in some set of circumstances (we call that the context) and is delivered to some other context. Value has many aspects, some of which cannot be made explicit. Meanwhile, context may change over time. Therefore there can no such thing as an objective or unchanging measure of quality. We can, however, discuss and construct a useful consensus about what value we think we're delivering.
Purpose of Testing	The purpose of testing is to detect non-conformances between a product and its specifications, so that they may be resolved. Specifications may exist on several levels, which leads to the concept of verification and validation. Verification means checking a component against its immediate spec, while validation means checking that it fulfills its ultimate requirements (that it is "fit for purpose").	The immediate purpose of testing is to understand the truth about the product. This in turn is done for other purposes. Usually the broader purpose is to find bugs. That means informing our clients about what *they* would consider to be anything about the product that threatens or unduly limits its value. In that case, a tester acts as an agent for people who have the power to decide what the product should be. Testing is sometimes done for other broad purposes, too, such as evaluating another testing process, training testers, or helping customer service prepare to support the product.
Central Questions of Testing	Does the product pass all the tests? Are there formal tests for each defined requirement?	Whom do we serve? What matters to them? Are we confident we know all the important problems in the product (regardless of defined requirements)? Without wasting our time or resources, is the testing adequate to detect every important problem that could reasonably be found (regardless of the formality of the testing)?
Unit of Work	The unit of testing work is usually a "test case," which may be a detailed set of instructions or a set of data for feeding to a formal (and perhaps automated) fact-checking mechanism. Often used interchangeably with the term "test."	There are no fixed "units of work" as such in RST. Testing is conceived as a deep intellectual process rather than an algorithmic mechanism. However, the central unit of concern in RST is the "test." A test is an experiment performed by a tester for the purposes of evaluating a product. Tests are usually embedded in a broader entity called a "test activity." Test activities may be structured in sessions (uninterrupted blocks of time), threads, or phases. Test sessions may be amenable to counting, and in Session-based Test Management they form a reasonably comparable unit of work that can be made visible to outsiders.

Method of Control	*Artifact-based and procedure-based management:* The process is intended to be manageable, ultimately, by non-testers or testers-not-present, using detailed instructions communicated explicitly via formal documents. These instructions (which may be loosely called "scripts") are followed by testers who are not expected to manage the value of their own time (but are expected to faithfully follow the instructions). Documents generally include: test plan, test case specifications (probably including test procedures as well), test results, occasionally a traceability matrix, too.	*People-based and activity-based management:* A tester managing a test process is called the "responsible tester" for that work. In RST it is important to trace who is responsible for each aspect of testing, and for that tester to be appropriately equipped and skilled in order to fulfill that role. Skilled testers manage their *own* processes (which may include personal supervision of unskilled testers) via a negotiated mission, formal and informal heuristics, and ongoing evaluation and communication of the emerging "testing story" comprised mainly of a description of test activities. Testing generally proceeds in an exploratory fashion, even though it may be formalized ("scripted") to some degree at the discretion of the responsible tester. Also, at the tester's discretion, many different forms of documents may be used to help manage the process. Documents are as concise as possible to minimize maintenance cost and maximize testing value. Typical documents include: risk outline, product coverage outline, test activity outline. These are often manifested as mindmaps or post-it notes. If high accountability and frequent formal reporting is needed, consider using thread-based or session-based test management to package and monitor test activities. Do not use metrics for any purpose of controlling people; use metrics only for purposes of casual inquiry, so as to provoke useful conversations. Another form of control is explicit heuristics, such as guidewords arranged in checklists that provide a reviewable structure to testing that helps when training, coordinating, or assessing groups of testers.
Approach to Estimating Work	Estimate required test cases based on review of specifications, if possible. Otherwise estimate by analogy to comparable projects.	Estimate work incrementally as you test or prepare to test, using the test estimation poster heuristic. Use all available information to identify necessary activities as well as any obstacles to the test process. Do not pretend to be able to predict how many bugs or builds or changes you will have to deal with—all of which may strongly impact testing. Avoid estimating if possible, but if needed, estimate test effort for an ideal (i.e. bug-free and instantly available) individual test cycle (i.e. testing needed for a single build) based on a requisite variety of test activities mapped to the full breadth of coverage areas. Express the estimation as a set of test session counts.
Approach to Test Design	Apply any of a list of named test techniques such as "equivalence class partitioning," or "boundary testing." Another approach is to itemize the parts of the specifications and write instructions that "try" or "tour" each of them. Whatever test design method is used, it should be standardized across the organization.	Test design is identical to experiment design as practiced in the sciences. All the methods and skills used in science are potentially relevant to software testing. We proceed by conjecture and refutation. Test design skill is gained through training and practice. Test techniques are heuristics that require skill, otherwise they are hollow. The basic method of test design is to model the product in a requisite variety of business-relevant ways, then determine ways to operate and interact with the product that "cover" the product with respect to those models while applying a requisite variety of business-relevant oracles to detect problems.

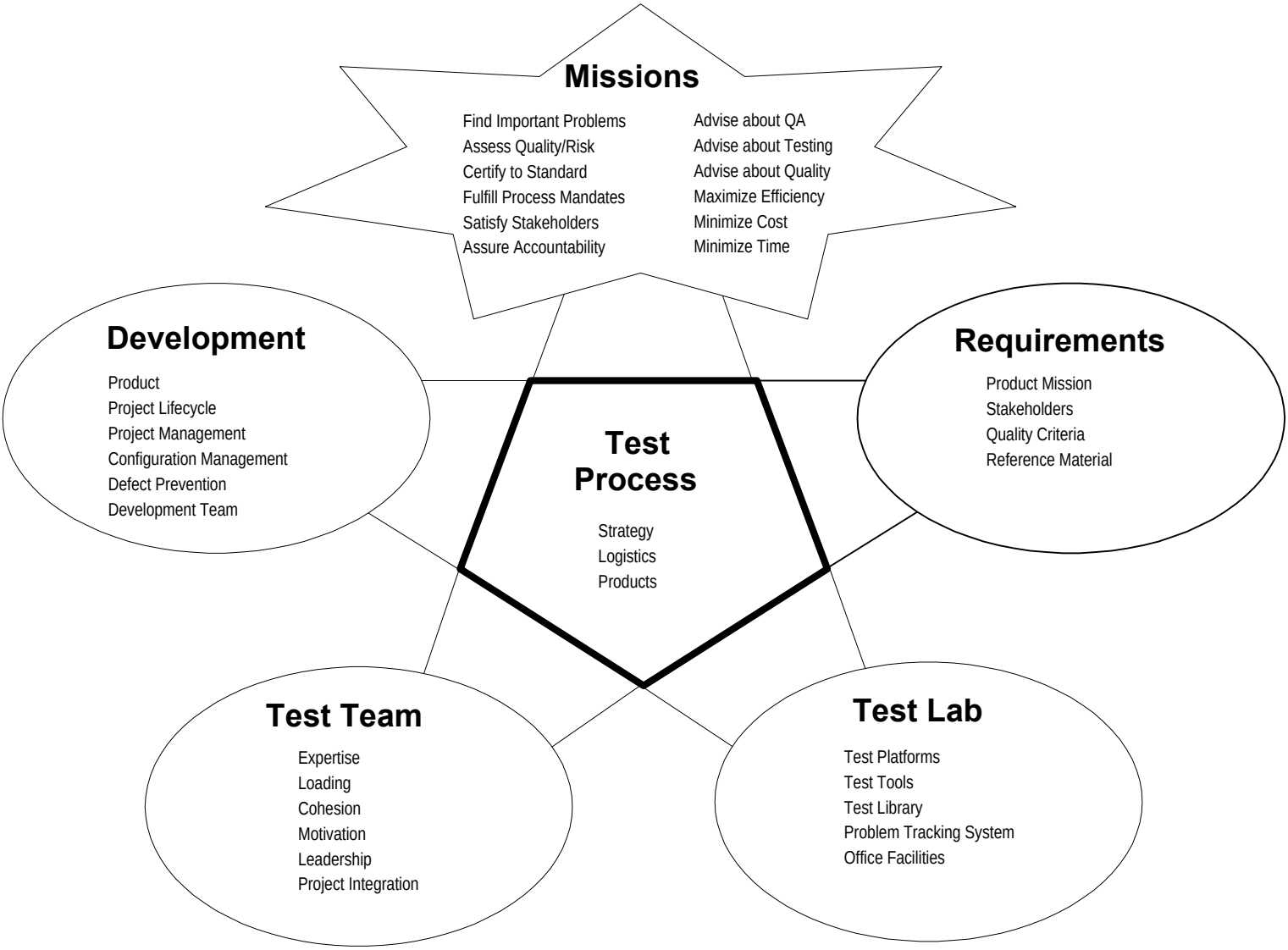
Ideal Sequence of Events	1. Receive correct specification. 2. Create test cases based on specification. 3. Set up the test lab and facilities. 4. (if possible) Automate test cases. 5. Receive product to test. 6. Run (or re-run) tests. 7. Report problems. 8. Receive new product with bug fixes. 9. Re-run tests and verify fixes.	There is no ideal sequence. Sequence and phasing of work is entirely contingent on the testing context. We ask ourselves "what's a good thing to do now?" Here are some general things that happen, which may happen in any order or portion, simultaneously or incrementally. • Learn about and model the product. • Analyze product risk. • Conceive of worthwhile test activities. • Test the product and report problems. • Explain and justify the testing. • Formalize the testing to improve test integrity. • Deformalize the testing to expand test coverage. • Set up the test lab and facilities. • Modify, redirect, and improve the work as conditions change. • Stop doing things that aren't helping enough. • Develop and improve relationships with the team. • Discover and experiment with new tools and methods.
Attitude Toward Change	V-Model or Waterfall. Prevent disruptive changes by proper planning in advance. Document the plan in detail, and prepare test plan and test cases in advance. Disciplined planning and communication eliminate surprises, later on.	We use an agile approach. We focus on preparation rather than planning. We focus on lowering the cost of exploratory cycles rather than deciding things up front. Whatever plans are made are going to change, so let's adapt to that change quickly.
Method of Assessing Testing	Review test artifacts. Review traceability of test cases to requirements. Capture and monitor metrics based on test case counts. Check whether documents conform to all process requirements. Possibly monitor code coverage using appropriate tools. Count bugs that escape the test process.	Discuss the testing with responsible tester. Personally observe testing (or demonstrations thereof). Observe and discuss the application of relevant heuristics. Evaluate the test strategy and test results relative to the needs of the business (we call that "test framing"). Don't count bugs that escape the test process (the count doesn't mean anything), instead investigate and learn from each one.
Definition of Done	All tests performed. Planned testing is complete. At this point the product is considered validated. The test results or report is then "signed off" by management.	All important questions about the status of the product have been answered. Clients are able to make well-informed decisions about it. Complete testing is impossible and there is no test for "always works." Instead we are obliged to stop when we conclude that further testing does not seem justified. Since we may be wrong about this, we evaluate the testing partly by the performance of the product in the field after the product is released. Since understanding of risk changes over the course of testing (testing is, in fact, an empirical form of risk analysis), we cannot rely on specific pre-specified "exit criteria" to decide when to stop. Furthermore, development activity constantly changes our baseline of understanding.
Role of Humans	Humans may play any of four roles in Factory School testing: designing methodologies, designing test procedures, automating test procedures, or following test procedures. Methodologists are rarely necessary. Instead, following perceived consensus standards and "best practices" is preferred. Test designers are not necessarily the same people who follow the test procedures that the designers create, but might be. Test design and execution are almost always two separate processes, however, regardless of whether they are done by the same people or different people. Automation is important, because tools are seen as a way to make test execution cheap and reliable.	The role of humans is central. There are three basic roles: test lead, responsible tester, and supporting tester. A responsible tester is a tester in charge of testing some part of a product, and is able to control his own methodology, procedures, tools, and activities. A test lead is a tester with three additional responsibilities: creating the conditions necessary for testing to succeed, coordinating the activities of other testers and helpers, and training testers. A supporting tester is someone who does testing activities under the supervision of a tester or lead, but is not responsible for the value of his own time. A supporting tester may be a senior person, such as an experienced developer, who is temporarily assisting the test process, or perhaps a novice tester not yet ready to take full responsibility.
Core Required Skill	The core skill is procedural discipline (in other words, the ability to follow instructions). Strongly relates to the ability to write instructions.	The core skill is ability to learn. This relates to curiosity, play, puzzle-solving, and tolerance for confusion.

Tester Diversity	Testers should be interchangeable. The test process benefits from standardization and formalization on all levels. Industry-wide certification makes it easier to find and foster appropriately qualified testers.	Each tester is unique, just as each lawyer, writer, or doctor is unique. Two testers may both be qualified to serve the same project, but we do not expect them to use the same methods or strategies, or perform the same tests in the same ways. For maximum effectiveness, a tester should work in a way that best exploits his own talents and temperament. The appropriate unit of analysis is the team, not the tester. Testing is served best by a diversified team—because that minimizes the probability of missing an important problem.
Role of Tacit Knowledge	There is no official role for tacit knowledge, although some techniques are defined as "experience-based" and job descriptions sometimes call for a certain number of years of experience, presumably because that may be correlated with higher competence of some unidentified kind.	Tacit knowledge is extremely important. The RST methodology is based on the premise that much of competence is tacit (unspoken) and is conveyed not through listening or reading to explicit instructions, but rather through observation of natural work, deliberative practical problem-solving, and live coaching by a supervisor. RST makes extensive and systematic use of heuristics that activate and direct tacit knowledge and skill.
Shifting Work to a New Tester	The role of humans should be minimized. In a well-run factory-style test process, it shouldn't matter who is doing the testing. The testing artifacts define the testing so that anyone can read them. Any new tester reads the documentation and follows the procedures. Automation should be used wherever possible to make this a moot point.	The role of humans is primary. Every tester is different. No one is interchangeable, even though all competent testers are potentially interoperable. Any skilled tester is capable of testing any product from scratch, to a reasonable degree, given reasonable time to prepare. Any unskilled tester will be working under supervision. If there are no skilled testers, then good testing will be impossible no matter what methodology you try. In any situation where testing is or should be formalized, records of some kind are typically produced. Concise notes, tables, or other artifacts—up to and including extremely detailed and rigorous test procedure documentation—may be created. A tester may use such material to take over testing from another tester. However, the receiving tester must be able to take full responsibility for the contents of what he inherits. Any mysterious document or tool must be discarded or recreated. Mysterious instructions are a potential hazard to the project. Testers may also pass work to each other through paired work, or through talking or live demonstration.
Role of Tools	Tools should be used to store and track testing documents and artifacts, as well as to automate test execution as much as possible. Tool use should be standardized across the organization.	In RST, we say testing cannot be automated, because any testing-like activity done exclusively by an algorithmic process is called "checking." We do this for the same reason that programmers call automated programming "compiling." It is important to distinguish between the capability and responsibility of humans vs. that of machines. However, testing may be supported and expanded by the use of tools. Testers and test teams are strongly encouraged to innovate and experiment with tools. Testers should develop or acquire any tools that might help make their testing more powerful or reliable, as long as these don't cost too much or create an unhelpful bias in test coverage. While it is not required or even desirable for every tester to be a programmer, a high functioning test team will have the ability to put tools in place quickly and inexpensively as the needs arise.

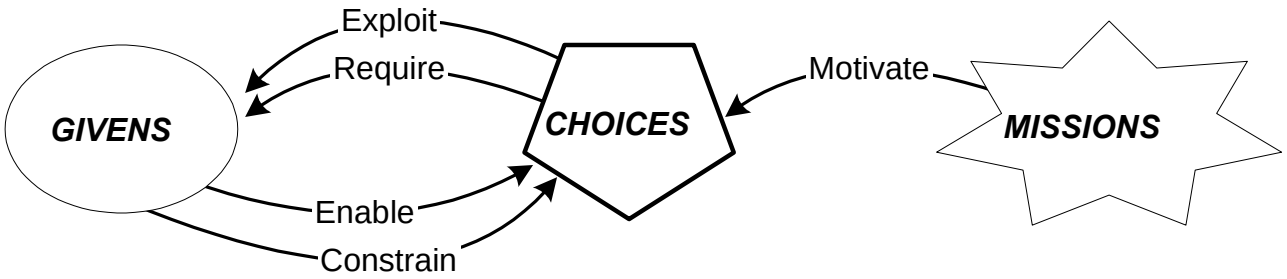
Heuristic Test Planning: Context Model

Designed by James Bach, http://www.satisfice.com

v1.2



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.

How To Evolve a Context-Driven Test Plan

This guide will assist you with your test planning. Remember, the real test plan is the set of ideas that actually guides your testing. We've designed the guide to be helpful whether or not you are writing a test plan *document*.

This is not a template. It's not a format to be "filled out." It's a set of ideas meant to jog your thinking, so you'll be less likely to forget something important. We use terse language and descriptions that may not be suited to a novice tester. It's designed more to support an experienced tester or test lead.

Below are seven task themes. Visit the themes in any order. In fact, jump freely from one to the other. Just realize that the quality of your test plan is related to how well you've performed tasks and considered issues like the ones documented below. The *Status Check* sections will help you decide when you have a good enough plan, but we recommend revisiting and revising your plan (at least in your head) throughout the project.

1. Monitor major test planning challenges.

Look for risks, roadblocks, or other challenges that will impact the time, effort, or feasibility of planning a practical and effective test strategy. Get a sense for the overall scope of the planning effort. Monitor these issues throughout the project.

Status Check

- ❑ Are any product quality standards especially critical to achieve or difficult to measure?
- ❑ Is the product especially complex or hard to learn? *(…then you will need more time)*
- ❑ Will testers require special training or tools? *(…then better go get that soon)*
- ❑ Are you remote from the users of the product? *(…then might have to work harder to understand them)*
- ❑ Are you remote from any of your clients? *(…then set up more disciplined process of communication)*
- ❑ Is any part of the test platform difficult to obtain or configure? *(…then warn your clients about that)*
- ❑ Will you test unintegrated or semi-operable product components? *(…then prep tools and environments)*
- ❑ Are there any particular testability problems? *(…then advocate for testability)*
- ❑ Does the project team lack experience with the product design, technology, or user base?
- ❑ Does test execution have to start soon? *(…then focus on what you really need to get started)*
- ❑ Is any information needed for planning not yet available? *(…then warn your clients you need it)*
- ❑ Are you unable to review a version of the product to be tested (even a demo, prototype, or old version)?
- ❑ Is adequate testing staff difficult to hire or organize? *(…then don't wait long before trying to do that)*
- ❑ Must you adhere to an unfamiliar test methodology? *(…then you better study it and see if you can)*
- ❑ Are you being pressured to formalize too soon? *(…then push back, because informality comes first)*
- ❑ Are project plans made without regard to testing needs? *(…then warn your clients that will impair testing)*
- ❑ Is the plan subject to lengthy negotiation or approval? *(…then start ASAP)*
- ❑ Are project plans changing frequently? *(…then establish a way to find out when they do)*
- ❑ Will the plan be subject to audit? *(…then find out what that audit process is and when it happens)*
- ❑ Are your clients unsure of what they want from you? *(…then tell them what they should want)*

2. Know your mission.

Any or all of the goals below may be part of your testing mission, and some more important than others. Based on your knowledge of the project, rank these goals. For any that apply, discover any specific success metrics by which you'll be judged.

Mission Elements to Consider

- ❑ Find important problems fast.
- ❑ Perform a comprehensive quality assessment.
- ❑ Certify product quality against a specific standard.
- ❑ Minimize testing time or cost.
- ❑ Maximize testing efficiency.
- ❑ Advise clients on improving quality or testability.
- ❑ Advise clients on how to test.
- ❑ Assure that the test process is fully accountable.
- ❑ Rigorously follow certain methods or instructions.
- ❑ Satisfy particular stakeholders.

Possible Work Products

- ❑ Brief email outlining your mission.
- ❑ One-page test project charter.

Status Check

- ❑ Do you know who your clients are?
- ❑ Do the people who matter agree on your mission?
- ❑ Is your mission sufficiently clear that you can base your planning on it?

3. Know the product.

Get to know the product and the underlying technology. Learn how the product will be used. Steep yourself in it. As you progress through the project, your testing will become better because you will be more of a product expert.

What to Analyze

- ❑ Users (who they are and what they do)
- ❑ Structure (code, files, etc.)
- ❑ Functions (what the product does)
- ❑ Data (input, output, states, etc.)
- ❑ Interfaces (user interfaces, APIs, connections to platform components)
- ❑ Platforms (external hardware and software)
- ❑ Operations (how product is used in real life)
- ❑ Timing (performance variables, periodic functionalities, race conditions)

Ways to Analyze

- ❑ Perform survey testing (testing with the primary goal of learning about the product).
- ❑ Apply the Heuristic Test Strategy Model product elements guidewords.
- ❑ Review product and project documentation.
- ❑ Interview designers and users.
- ❑ Compare w/similar products.

Possible Work Products

- ❑ Product coverage outline
- ❑ Annotated specifications
- ❑ Product bug list
- ❑ Project issue list

Status Check

- ❑ Do designers approve of the product coverage outline?
- ❑ Do designers think you understand the product?
- ❑ Can you visualize the product and predict behavior?
- ❑ Are you able to produce test data (input and results)?
- ❑ Can you configure and operate the product?
- ❑ Do you understand how the product will be used?
- ❑ Are you aware of gaps or inconsistencies in the design?
- ❑ Have you considered tacit specifications as well as explicit?

4. Know the risk.

How might this product fail in a way that matters? At first you'll have a general idea, at best. As you progress through the project, your test strategy, your testing will become better because you'll learn more about the failure dynamics of the product.

What to Analyze

- ❑ Threats (challenging situations and data)
- ❑ Vulnerabilities (where it's likely to fail)
- ❑ Failure modes (possible kinds of problems)
- ❑ Victim impact (how problems matter)

Ways to Analyze

- ❑ Perform survey testing or general shallow testing to identify risk hotspots.
- ❑ Review product against risk heuristics and quality criteria categories.
- ❑ Review requirements and specifications.
- ❑ Review actual failures in the lab or in the field.
- ❑ Review code and architecture to understand failure modes and fault propagation pathways.
- ❑ Interview designers and users.
- ❑ Use a risk catalog to identify problems you want to specifically test for.

Possible Work Products

- ❑ Component/risk matrix (outline of the parts of the product and risk factors associated with them)
- ❑ List of risk areas (clusters of related suspected risks)
- ❑ List of risk factors (threats and vulnerabilities)
- ❑ Risk catalog (outline of all the kinds of problems that typically occur with that technology)

Status Check

- ❑ Do the designers and users concur with the risk analysis?
- ❑ Will you be able to detect all significant kinds of problems, should they occur during testing?
- ❑ Do you know where to focus testing effort for maximum effectiveness?
- ❑ Can the designers do anything to make important problems easier to detect, or less likely to occur?
- ❑ What makes you think your risk analysis is any good? Have a compelling story about that.

5. Decide the test strategy.

What can you do to test rapidly and effectively based on the best information you have about the product? Make the best decisions you can, up front, but let your strategy improve throughout the project.

Consider Techniques From Five Perspectives

- ❑ Tester-focused techniques.
- ❑ Coverage-focused techniques (both structural and functional).
- ❑ Problem-focused techniques.
- ❑ Activity-focused techniques.
- ❑ Oracle-focused techniques.

Ways to Plan

- ❑ Match techniques to risks and product areas.
- ❑ Visualize specific and practical techniques.
- ❑ Diversify your strategy to minimize the chance of missing important problems.
- ❑ Look for ways automation could allow you to expand your strategy.
- ❑ Don't overplan. Testers must stay awake by using their brains.

Possible Work Products

- ❑ Itemized statement of each test strategy chosen and how it will be applied.
- ❑ Risk/task matrix.
- ❑ List of issues or challenges inherent in the chosen strategies.
- ❑ Advisory of poorly covered parts of the product.
- ❑ Test cases (only if required)

Status Check

- ❑ Do your clients concur with the test strategy?
- ❑ Is everything in the test strategy necessary?
- ❑ Can you actually carry out this strategy?
- ❑ Is the test strategy too generic—could it just as easily apply to any product?
- ❑ Is there any category of important problem that you know you are not testing for?
- ❑ Has the strategy made use of available resources and helpers?

6. Know the logistics.

How will you implement your strategy? Your test strategy is profoundly affected by logistical constraints or mandates. Try to negotiate for the resources you need and exploit whatever you have.

Logistical Areas

- ❑ Making contact with users
- ❑ Making contact with your clients
- ❑ Test effort estimation and scheduling
- ❑ Testability advocacy
- ❑ Test team staffing (right skills)
- ❑ Tester training and supervision
- ❑ Tester task assignments
- ❑ Product information gathering and management
- ❑ Project meetings, communication, and coordination
- ❑ Relations with all other project functions, including development
- ❑ Test platform acquisition and configuration
- ❑ Agreements and protocols
- ❑ Test tools and automation
- ❑ Stubbing and simulation needs
- ❑ Test documentation management and maintenance
- ❑ Build and transmittal protocol
- ❑ Test cycle administration, especially after changes
- ❑ Bug reporting system and protocol
- ❑ Test status reporting protocol
- ❑ Code freeze and incremental testing
- ❑ Pressure management in the end game
- ❑ Sign-off protocol
- ❑ Evaluation of test effectiveness, including escaped bug analysis

Possible Work Products

- ❑ Project agreements
- ❑ Project issues list
- ❑ Project risk analysis
- ❑ Responsibility matrix
- ❑ Test schedule

Status Check

- ❑ Do the logistics of the project support the test strategy?
- ❑ Are there any problems that block testing?
- ❑ Are the logistics and strategy adaptable in the face of foreseeable problems?
- ❑ Can you start testing now and sort out the rest of the issues later?

7. Share the plan.

You are not alone. The test process must serve the project. So, involve the project in your test planning process. You don't have to be grandiose about it. At least chat with key members of the team to get their perspective and implicit consent to pursue your plan.

Ways to Share

- ❑ Engage designers and stakeholders in the test planning process.
- ❑ Actively solicit opinions about the test plan.
- ❑ Do everything possible to help the developers succeed.
- ❑ Help the developers understand how what they do impacts testing.
- ❑ Talk to technical writers and technical support people about sharing quality information.
- ❑ Get designers and developers to review and approve reference materials.
- ❑ Record and track agreements.
- ❑ Get people to review the plan in pieces.
- ❑ Improve reviewability by minimizing unnecessary text in test plan documents.

Goals

- ❑ Common understanding of the test process.
- ❑ Common commitment to the test process.
- ❑ Reasonable participation in the test process.
- ❑ Management has reasonable expectations about the test process.

Status Check

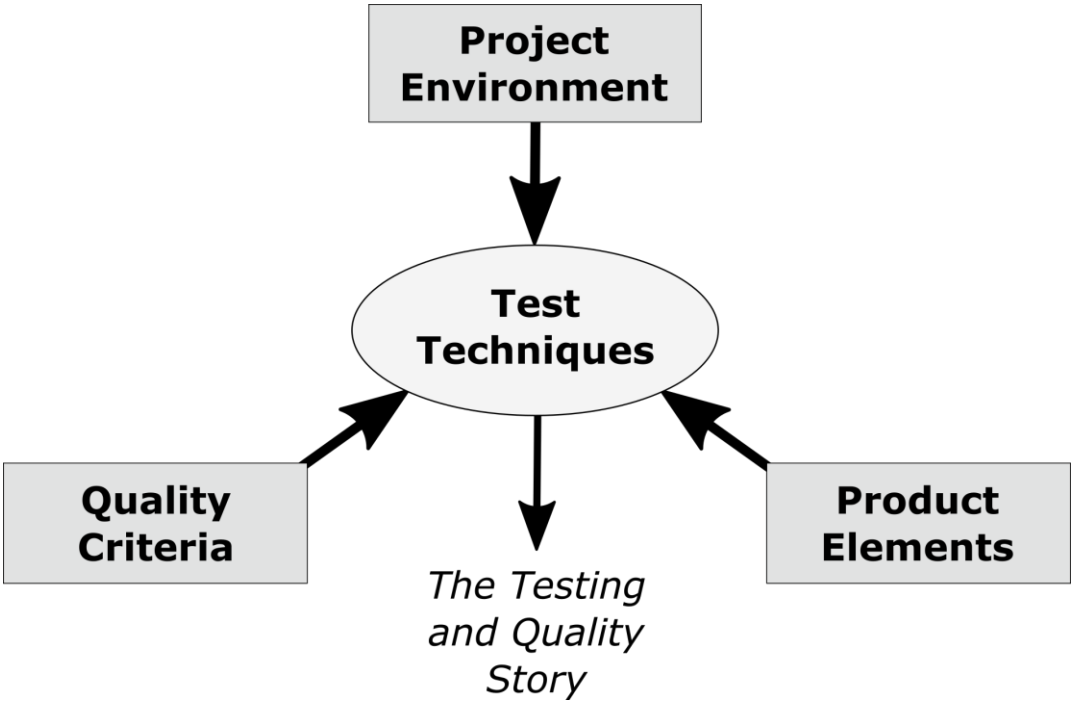
- ❑ Is the project team paying attention to the test plan?
- ❑ Does the project team, especially first line management, understand the role of the test team?
- ❑ Does the project team feel that the test team has the best interests of the project at heart?
- ❑ Is there an adversarial or constructive relationship between the test team and the rest of the project?
- ❑ Does anyone feel that the testers are "off on a tangent" rather than focused on important testing?

Designed by James Bach
james@satisfice.com
www.satisfice.com

Version 6.0
2/8/2024

Heuristic Test Strategy Model

The **Heuristic Test Strategy Model** is a set of patterns for designing and choosing tests to perform. The immediate purpose of this model is to remind testers of what to think about during that process. I encourage testers to customize it to fit their own organizations and contexts.



Project Environment represents a set of context factors that include resources, constraints, and other elements in the project that may enable or hobble our testing. Sometimes a tester must challenge constraints, and sometimes accept them.

Product Elements are aspects of the product that you consider testing, including aspects intrinsic to the product and relationships between the product and things outside it. Software is complex and invisible. Take care to cover all of it that matters, not just the parts that are easy to see.

Quality Criteria Categories are dimensions in which people determine the value of the product. You can also think of them as categories of product risk. Quality criteria are subjective and multidimensional and often hidden or contradictory.

General Test Techniques are heuristics for designing tests. How, where, and when to apply a particular technique requires an analysis of project environment, product elements, and quality criteria.

The Testing and Quality Story is the result of testing. You can never know the "actual" quality of a software product— you can't "verify" quality, as such— but by performing tests, you can make an assessment, and that takes the form of a story you tell (including bugs, curios, etc.).

General Test Techniques

A test technique is a heuristic for designing tests. There are many interesting techniques. The list includes nine families of general techniques. By "general technique" we 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 families. And an endless variety of specific test techniques may be constructed by combining one or more general techniques with coverage ideas from the other lists in this 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

Partition 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.
4. Consider using inputs that force the whole range of possible outputs to occur.

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. Perform multiple activities connected end-to-end; for instance, conduct tours through a state model.
2. Don't reset the system between actions.
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

Challenge every claim

1. Identify reference materials that include claims about the product (tacit or explicit). Consider SLAs, EULAs, advertisements, specifications, help text, manuals, etc.
2. Analyze individual claims, and clarify vague claims.
3. Test each claim about the product.
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, logs based on user activity, 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.

Tool-Supported Testing

Use tools to make testers more powerful

1. Look for or develop tools that can perform a lot of actions and check a lot of things.
2. Consider tools that partially automate test coverage.
3. Consider tools that partially automate oracles.
4. Consider automatic change detectors.
5. Consider automatic test data generators.

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 specific tests to create. In each category, below, consider how that element may help or hinder your test design process. Try to exploit every resource.

Mission. *Your purpose on this project, as understood by you and your customers.*

- Why are you testing? Are you motivated by a general concern about quality or specific and defined risks?
- Do you know who the customers of your work are? Whose opinions matter? Who benefits or suffers from the work you do?
- Maybe the people you serve have strong ideas about what tests you should create and run. Find out.
- Have you negotiated project conditions that affect your ability to accept your mission?

Information. *Information about the product or project that is needed for testing.*

- Whom can we consult with to learn about this project?
- Are there any engineering documents available? User manuals? Web-based materials? Specs? User stories?
- Does this product have a history? Old problems that were fixed or deferred? Pattern of customer complaints?
- Is your information current? How are you apprised of new or changing information?
- Are there any comparable products or projects from which we can glean important information?

Developer Relations. *How you get along with the programmers.*

- *Rapport:* Have you developed a friendly working relationship 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?
- *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? Do they have the knowledge and skills they need?
- 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?
- Are there particular test techniques that someone on the team has special skill or motivation to perform?
- Who is co-located and who is elsewhere? Will time zones be a problem?

Equipment & Tools. *Hardware, software, or documents required to administer testing.*

- *Hardware:* Do you have all the physical or virtual hardware you need for testing? Do you control it or share it?
- *Automated Checking*: Do you have tools that allow you to control and observe product behavior automatically?
- *Analytical Tools*: Do you have tools to create test data, design scenarios, or to analyze and track test results?
- *Matrices & Checklists*: Are any documents needed to track or record the progress of testing?
- *Signals:* Do you have access to engineering data coming back from the field?

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 performed? Are some tests performed 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? Do you have test platforms available? Will you test in production?
- *Interoperable Systems:* Are any third-party services required for your product that must be mocked or made available?
- *Volatility:* Is the product constantly changing? How will you find out about changes?
- *New Stuff:* Do you know 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 testing, 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. Each category, listed below, represents an important and unique element to be considered in the test strategy. 20

Structure. *Everything that comprises the physical product.*

- *Code:* the code structures that comprise the product, from executables to individual routines.
- *Hardware:* any hardware component that is integral to the product.
- *Service:* any server or process running independently of others that may comprise the product.
- *Non-executable files:* any files other than multimedia or programs, like text files, sample data, or help files.
- *Collateral:* anything beyond that is also part of the product, such as paper documents, web pages, packaging, license agreements, etc.

Function. *Everything that the product does.*

- *Multi-user/Social:* any function designed to facilitate interaction among people or to allow concurrent access to the same resources.
- *Calculation:* any arithmetic function or arithmetic operations embedded in other functions.
- *Time-related:* time-out settings; periodic events; time zones; business holidays; terms and warranty periods; chronograph functions.
- *Security-related:* rights of each class of user; protection of data; encryption; front end vs. back end protections; vulnerabilities in sub-systems.
- *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 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 and produces.*

- *Input/Output:* any data that is processed by the product, and any data that results from that processing.
- *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 expected to persist over multiple operations. This includes modes or states of the product, such as options settings, view modes, contents of documents, etc.
- *Interdependent/Interacting:* any data that influences or is influenced by the state of other data; or jointly influences an output.
- *Sequences/Combinations:* any ordering or permutation of data, e.g. word order, sorted vs. unsorted data, order of tests.
- *Cardinality:* numbers of objects or fields may vary (e.g. zero, one, many, max, open limit). Some may have to be unique (e.g. database keys).
- *Big/Little:* variations in the size and aggregation of data.
- *Invalid/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.

Interfaces. *Every conduit by which the product is accessed or expressed.*

- *User Interfaces:* any element that mediates the exchange of data with the user (e.g. displays, buttons, fields, whether physical or virtual).
- *System Interfaces:* any interface with something other than a user, such as engineering logs, other programs, hard disk, network, etc.
- *API/SDK:* any programmatic interfaces or tools intended to allow the development of new applications using this product.
- *Import/export:* any functions that package data for use by a different product, or interpret data from a different product.

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) for the product to work: systems, servers, memory, keyboards, the Cloud.
- *External Software:* software components and configurations that are not a part of the shipping product, but are required (or optional) for the product to work: operating systems, concurrently executing applications, drivers, fonts, etc.
- *Embedded Components:* libraries and other components that are embedded in your product but are produced outside your project.
- *Product Footprint:* The resources in the environment that are used, reserved, or consumed by the product (memory, filehandles, etc.)

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 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.

Capability. *Can it perform the required functions?*

- *Sufficiency:* the product possesses all the capabilities necessary to serve its purpose.
- *Correctness:* it is possible for the product to function as intended and produce acceptable output.

Reliability. *Will it work well and resist failure in all required situations?*

- *Robustness:* the product continues to function over time without degradation, under reasonable conditions.
- *Error handling:* the product resists failure in the case of bad data, 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.

Charisma. *How appealing is the product?*

- *Aesthetics:* the product appeals to the senses.
- *Uniqueness:* the product is new or special in some way.
- *Entrancement:* users get hooked, have fun, are fully engaged when using the product.
- *Image:* the product projects the desired impression of quality.

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 he says he 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?*

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.
- *Product Footprint:* the product doesn't unnecessarily hog memory, storage, or other system resources.

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/patches:* Can new modules or versions be added easily? Do they respect the existing configuration?
- *Administration:* Is installation a process that is handled by special personnel, or on a special schedule?

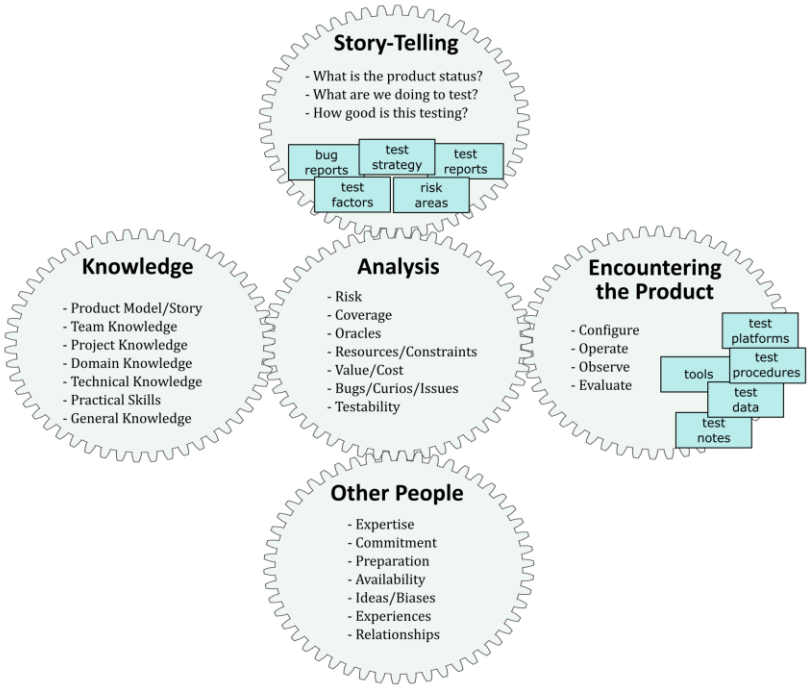
Development. *How well can we create, test, and modify it?*

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

Elements of Excellent Testing

Created by James Bach, Jonathan Bach, and Michael Bolton *v4.2*

Science is testing; and *testing is science*. The world of commercial product testing differs from the world of science mainly in its object, not its subject or its process: we who test products apply ourselves to the study of an ephemeral human contrivance rather than the natural world. This is a human learning process.



Testing, like science, is an exploratory process that also makes use of scripted elements. The "gears" in the diagram to the left represent activities that evolve over time, feeding each other.

The rectangles are artifacts that result from those activities.

At the center, analysis drives the whole process. The four connection points to analysis are worth examining closer.

Learning: The connections 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 connection between analysis and experiment might be called the testing loop (in the sense of performing tests). It is dominated by questions about the status of the product. It may involve algorithmic processes such as automated output checking.

Collaboration: The connections between analysis and other people might be called the collaboration loop. Testing is always a social process to some degree, and group testing can be very energizing.

Self-management: The connection between analysis and the testing story is self-management, by which the whole process is regulated. Self-management is driven by stories we tell ourselves.

Evolving Work Products

As testing proceeds, look for any of the following to be created, refined, and possibly documented during the process.

	Test Ideas. Any idea or part of an idea, written or unwritten, that may guide the performance of a test or check.
	Output Checks. Mechanized or mechanizable processes for gathering product observations and evaluating them. A test is always human-guided, whereas a check, by definition, can be completely automated. A test often includes one or more checks, but a check cannot include a test.
	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. Reports regarding anything about the *product* that threatens its value.
	Issue Reports. Reports regarding anything about the *project* that threatens its value.
	Test Conditions (Product Coverage Outline). A *test condition* is anything about the product we might want to examine with a test. A *product coverage outline* is an outline, list or table of interesting test conditions.
	Product Risks. Any potential areas of bugginess or types of bug.
	Test Data. Any data available for use in testing.
	Test Tools. Any tools acquired or developed to aid testing (includes automated output checks).
	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 and what obstacles might be in our way.
	Testing Story. What we know about our testing, so far.
	Product Story. What we know about the status of the product, so far.
	Test Process Assessment. Our own assessment of the quality of our test process.
	Tester and Team. The tester and the team evolves over the course of the project.
	Technical and Domain Knowledge. Our knowledge about how the product works and how it is used.

Testing Skills and Tactics

These are the skills (or tactics that involve skill) needed for professional and cost effective product testing. Each is distinctly observable and learnable, and each is necessary for excellent exploratory work.

Self-Management Skills and Tactics

	Chartering your work. Making decisions about what you will work on and how you will work. Deciding the testing story you want to manifest; knowing 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.
	Self-care. Monitoring your emotional, physical, and mental states as they influence your testing; taking effective action to manage your energy and maintain a positive outlook; self-forgiveness.
	Self-criticism. Finding problems in your work and correcting them; awareness and acknowledgement of your strengths and weaknesses as a tester.
	Test status evaluation. Maintaining an awareness of problems, obstacles, limitations and biases in your testing; understanding the cost vs. value of the work; constructing the testing story.
	Ethics. Understanding your ethical code and fulfilling your responsibilities under it as you 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; optimizing effort; using focusing heuristics.
	De-focusing your work. Expanding the scope of 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 complementary 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.
	Knowing 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.
	Developing and maintaining credibility. No one will listen to you if they think what you say is not interesting or important. Remember, a tester has very little visible work, so your reputation is paramount.

Collaboration Skills and Tactics

	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.
	Conversation. Talking through and elaborating ideas with other people.
	Serving other testers. Performing services that support other testers on their own terms.
	Guiding other testers. Supervising testers who support your explorations; coaching testers.
	Asking for help. Articulating your needs; negotiating for assistance.
	Role visiting. Where feasible and applicable, spending time performing non-testing roles that may give you perspective or practice that makes you a better tester.
	Telling the story of your testing. 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 product you are studying, including bugs found. This is the ultimate goal for most test projects.

Learning Skills and Tactics

	Discovering and developing resources. Obtaining information or facilities to support your effort. Exploring those resources.
	Using 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 or Wikipedia.
	Considering history. Reviewing what's been done before and mining that resource for better ideas.
	Reading and analyzing documents. Reading carefully and analyzing the logic and ideas within documents that pertain to your subject.
	Interviewing. 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.

Test Performance Skills and Tactics

	Encountering the product. Making and managing contact with the subject of your study; for technology, configuring and operating it so that it demonstrates what it can do.
	Sensemaking. Determining the meaning and significance of what you encounter; considering multiple, incompatible explanations that account for the same facts; inference to the best explanation.
	Modeling and factoring. Modeling means composing, decomposing, describing, and working with mental representations of the things you are exploring; factoring means identifying relevant dimensions, variables, and dynamics that should be tested. There are lots of formal modeling methods, as well.
	Analyzing product risk. Using experiential data, conversation, and heuristics, identify suspected product risks that deserve to be investigated with testing.
	Experiment design. 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.
	Literate observation. Making *relevant* observations guided by your various mental models; gathering different kinds of empirical data, or data about different aspects of the object; establishing procedures for rigorous observations; noticing strange things; noticing what you are *not* seeing.
	Detecting potential problems. Designing and applying oracles to detect behaviors and attributes that may be trouble.
	Assessing validity. Analyzing, monitoring, and correcting for factors that may distort or invalidate the tests.
	Notetaking. Recording observations, ideas, and progress as you test; recording useful information without unduly disturbing the test process itself.
	Data wrangling. Synthesizing, modifying, moving, and reformatting test data.
	Bug reporting and advocacy. Explaining problems in a compelling and respectful way.
	Applying tools. Enabling new kinds of work or improving existing work by developing and deploying tools.
	Testability advocacy. Analyzing and negotiating for the conditions that make testing easier and more effective.
	Protocol design. Creating and following procedures and practices that increase the reliability of the test process.
	Lab management. Creating and maintaining the systems, tools, databases, and spaces that you need to test well.

Knowledge that Helps

In addition to the skills and tactics of testing, there's lots of things we might need to know.

	Product Knowledge. What does your product do and how does it work?
	Technology Knowledge. What technology is your product built from? What other technologies can help you test it?
	Project Knowledge. What's going on in your project? What's the schedule? Who is working on it?
	Domain Knowledge. Who are the users? How do they think? What sort of process does your product support?
	General systems knowledge. This refers to the whole field of general systems theory and systems thinking. In short, it consists of the heuristics and know-how about how dynamic systems behave.
	Tool Knowledge. Physical or software-based tools that can help testing. This does not only mean tools that are called "test tools" but rather ANY tool that may help ANY aspect of the test process.
	Test Technique Knowledge. There are many kinds of testing; many specific testing heuristics you might use.
	Resource Knowledge. In addition to things that you think of as tools, you need to be aware of any resource (i.e. facility, material, or service) that is available to help you get the job done.
	People Knowledge. Who can help you? What skills do they have that you need? How do you approach them? How specifically might they contribute to the test project?
	Role Knowledge. What do the other people do who make the project work? How does their work impact yours? How might knowing more about their roles help you do your job better? How do you serve them as a tester?
	History Knowledge. What is the history of this project? This product line? The market? This company? What trouble has happened in the past that we don't want repeated?
	Business and Market Knowledge. Who are your competitors? What are those competing products? Are there similar or complementary products as well? How does quality affect your bottom line?

Helpful Skills Some Testers Have

In addition to the defining skills of testing, there are other skills and knowledge areas that testers may have.

	Coding Skill. In some companies, coding skills are a requirement. In general, the ability to build your own tools brings great power to testing. However, people with coding skills may also think too much like coders and lose empathy for users.
	Design Skill. Product design, and especially user interface design, can help sharpen your bug reports and improve your bug detection ability.
	Social Science Skills. The social sciences are about studying extremely complex, socially situated phenomena. The analytical methods and standards of social science helps testers better understand the limits of software testing and to better study and improve testing processes.
	Specification Writing Skills. Sometimes it helps for testers to help write specifications, or to suggest rewrites. Writing a spec is one great way of preparing to design tests for that product.
	Mathematics and Logic Skills. Statistics, combinatorics, and formal logic are often useful to design deep tests and characterize test coverage.
	Cognitive Science Skills. If you understand the patterns and limitations of human perception, you can better appreciate how to avoid common pitfalls of self-deception, and to design test procedures that are more reliable.

Exploratory 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
	Deliberate vs. spontaneous
	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.
- ❑ *Interfaces.* Every conduit by which the product is accessed or expressed.
- ❑ *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?
- ❑ *Charisma.* How appealing is 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?
- ❑ *Compatibility.* How well does it work with external components & configurations?
- ❑ *Performance.* How speedy and responsive is it?
- ❑ *Installability.* How easily can it be installed onto it target platform?
- ❑ *Development.* How well can we create, test, and modify it?

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.

The Role/Actor Heuristic (v1.1)

Dimensions of Role

Dimension	Typical Problems	Typical Remedies
Scope (what the role covers) • Responsibilities • What depends on it • What it depends on	• Role too big for actor; tasks get lost • Big role shared by many actors who fight each other	• Bring more actors in to share role • Break up big role into smaller roles • Create manager role
Power (what the role influences) • Authority/Sponsorship • What roles control it • What roles it controls	• Powerful role leaves others without enough power • Weak role can't get what it needs • Weak role controlled by strong role that doesn't understand it	• Break up big role into smaller roles • Redistribute power • Attach weak role to stronger roles • Strengthen role via stronger actor • Educate controlling roles • Create manager role
Value (what the role does for people) • Specific problems solved • Necessity to organization • Desirability to others • Prestige for actor • New problems created	• Role is not important enough; wastes time and effort • Role creates too much trouble • Role is unpopular and is undermined • Role is thankless and no one wants it • Role ruined by bad actors	• Ritualize or eliminate role • Increase power of role • Get better actors • Use "role model" as actor
Cost (what the role takes from people) • Cost of the actor, equipment, and materials • Cost to accommodate the role • Cost due to other roles becoming complacent	• Role is too expensive • Role makes costs uncomfortably public • Existence of a role causes others to "leave it to the expert" and lose skill. • Necessity to accommodate the role disrupts other roles	• Eliminate role in order to hide cost • Hire highly skilled actors • Hire extremely inexpensive actors • Promote the value of the role to show that costs are justified
Requirements (what role/actor needs) • Environment & tools • Skills & knowledge • Motivation • Outside support	• Requirements are too hard to fulfill • Qualified actors are too hard to recruit • Weak role can't get what it needs	• Make do with less and communicate impact to sponsor • Offer training and coaching • Increase prestige of role • Ritualize or eliminate role
Openness (how actors relate to it) • Ownership & commitment • Casual shareability • Informality • Interruptability • Simplicity • Legibility	• Role is highly territorial • Role is easily disrupted by helpers • Role is difficult to adopt • Role is difficult to let go of • Role is mysterious and opaque • Role too reliant on specific actors • Role is a tragic commons	• Ritualize or eliminate role • Offer training and coaching • Strengthen role • Make strong agreements with actors • Close the role to outsiders • Formalize to improve legibility
Presence (when & where it operates) • Persistence • Responsiveness • Disruptiveness	• Response is too slow • When role goes away and later comes back, people forget many details • Role slows down other roles	• Add more actors to speed it up • Good documentation to preserve history • Ritualize or eliminate the role

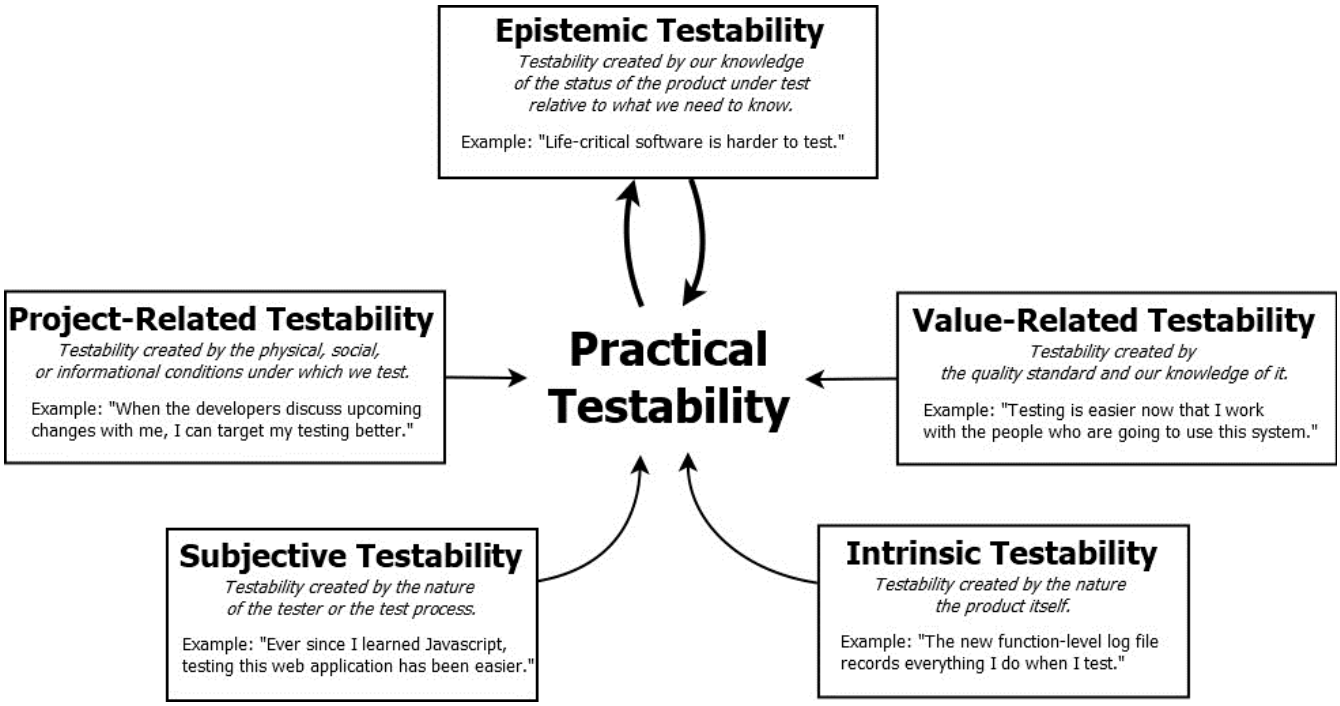
Expectations of Actors

Expectation	Typical Problems	Typical Remedies
Commitment (acceptance of duty) • Investment of energy • Accountability	• Conflict of commitment between projects • It may not be clear who to blame • There may be many causes for a problem	• Do fewer projects • Persuade people to commit • Management must watch and listen
Competence (ability to perform) • Study and practice • Self-evaluation	• No training available • Training is actively harmful • Dunning-Krueger syndrome	• On-the-job coaching • Personal ambition • Fake it and hope no one fires you
Readiness (operational status) • Anticipating events • Adapting to new conditions • Maintaining efficiency • Troubleshooting	• Chronically unanticipated obstacles • Black swan obstacle • Another role spoils your plans	• Hire a competent actor • Use a checklist or guideword heuristics • Readiness review • Increase power of role
Coordination (relating to other roles) • Mission negotiation • Resource negotiation • Helping and accepting help • Respecting agreements • Failover strategy • Status reporting • Delivery	• Goal displacement • Forgotten agreements • Partial delivery • Fail to justify need for resources • Poor credibility in negotiation	• Good meetings • Use a checklist or guideword heuristics • Use a map of dependencies • Better sponsorship • Increase power of role

Heuristics of Software Testability

Version 2.6, 2024, by James Bach, Satisfice, Inc.

The *practical testability* of a product is how easy it is to test* by a particular tester and test process, in a given context†. Practical testability is a function of five other "testabilities:" *project-related* testability, *value-related* testability, *subjective* testability, *intrinsic* testability, and *epistemic* testability (also known as the "risk gap"). Just as in the case for quality in general, testability is a plastic and multi-dimensional concept that cannot be usefully expressed in any single metric. But we can identify testability problems and heuristics for improving testability in general.



Interesting Testability Dynamics

Changing the product or raising the quality standard reduces epistemic testability. The difference between what we know and what we need to know is why we test in the first place. A product is easier to test if we already know a lot about its quality or if the quality standard is low, because there isn't much left for testing to do. That's epistemic testability. Therefore, product that changes a lot or in which we can't tolerate trouble is automatically less testable.

Improving any other aspect of testability increases the rate of improvement of epistemic testability. Efforts made to improve any other aspect of testability, by definition, increases the rate at which the gap between what we know and what we need to know closes.

Improving test strategy might *decrease* subjective testability or vice versa. This may happen when we realize that our existing way of testing, although easy to perform, is not working. A better test strategy, however, may require much more effort and skill. (Ignorance was bliss.) Beware that the opposite may also occur. We might make a change (adding a tool for instance) that makes testing seem easier, when in fact the testing is worse. (Bliss may be ignorant.)

---

* *Testing* is evaluating a product by learning about it through experiencing, exploring, and experimenting.

† *Context* is all of the factors a situation that should be considered when solving a situated problem.

Increasing intrinsic testability might *decrease* project-related testability. This may happen if redesigning a product to make it more testable also introduces many new bugs. Or it may happen because the developers spend longer stabilizing the product before letting independent testers see it. Agile development done well helps minimize this problem.

Increasing value-related testability might *decrease* project testability. You can dramatically increase value-related testability by embedding yourself with real users in the field, and even participating in their work. You will be testing under the most realistic possible conditions. However, this can be difficult, expensive, slow, and lead to accidental retention or mishandling confidential personal data as you test, which hurts project-related testability.

Increasing practical testability *also* improves development and maintenance. If a product is easier to test then it is also easier to support, debug, and evolve. Observability and controllability, for instance, is a tide that floats all boats.

The tester must *ask* for testability. We cannot expect any non-tester to seriously consider testability. It's nice when they do, but don't count on it. An excellent tester learns to spot testability issues and resolve them with the team.

Guidewords for Analyzing Testability

Epistemic Testability

- **Prior Knowledge of Quality.** If we already know a lot about a product, we don't need to do as much testing.
- **Tolerance for Failure.** The less quality required, or the more risk that can be taken, the less testing is needed.

Project-Related Testability

- **Supportive Culture.** Testers are highly sensitive to intimidation or disrespect. No one wants to feel like a troublemaker or outsider. Protect the independence and purpose of testers if you want their best work.
- **Developer Availabilty**. The ability to speak with and perhaps influence developers/designers/managers makes a nurturing environment for testing.
- **Change Control.** Frequent and disruptive change requires retesting and invalidates our existing product knowledge. Change control generally improves testability, but can also hurt it if we aren't allowed to update our test environments, tools, and data. Change control may occur on a local or corporate level.
- **Information Availability.** We get all information we want or need to test well.
- **Tool Availability.** We are provided all tools we want or need to test well.
- **Test Item Availability.** We can access and interact with all relevant versions of the product.
- **Sandboxing.** We are free to do any testing worth doing (perhaps including mutation or destructive testing), without fear of disrupting users, other testers, or the development process.
- **Environmental Controllability.** We can control all potentially relevant experimental variables in the environment surrounding our tests.
- **Time.** Having too little time destroys testability. We require time to think, prepare, and cope with surprises.
- **Leanness.** Complexity and magnitude of work products, accumulated over time, must be navigated or maintained to get the testing done. Also, complicated bureaucratic processes reduce time available for tsting. Avoid technical debt and administrative overhead.

Value-Related Testability

- **Oracle Availability.** We need ways to detect each kind of problem. A well-written specification is one example of such an oracle, but there are lots of other kinds of oracles (including people and tools).
- **Oracle Authority.** We benefit from oracles that identify problems that will be considered important.
- **Oracle Reliability.** We benefit from oracles that can be trusted to work over time and in many conditions.
- **Oracle Precision.** We benefit from oracles that facilitate identification of specific problems.
- **Oracle Inexpensiveness.** We benefit from oracles that don't require much cost or effort to acquire or operate.
- **User Stability & Unity.** The less users change and the more harmony among them, the easier the testing.
- **User Familiarity.** The more we understand and identify with users, the easier it is to test for them.

- **User Availability.** The more we can talk to and observe users, the easier it is to test for them.
- **User Data Availability.** The more access we have to natural data, the easier it is to test.
- **User Environment Availability.** Access to natural usage environments improves testing.
- **User Environment Stability & Unity.** The less user environments and platforms change and the fewer of them there are, the easier it is to test.

Subjective Testability

- **Engagement.** A bored or distressed tester is a poor tester. A good tester is intellectually and emotionally engaged in the work; confident of finding any problems that may exist.
- **Involvement.** Testing is easier when a tester is closer to the development process, communicating and collaborating well with the rest of the team. When testers are held away from development, test efficiency suffers terribly.
- **Product Knowledge.** Knowing a lot about the product, including how it works internally, profoundly improves our ability to test it. If we don't know about the product, testing with an exploratory approach helps us to learn rapidly.
- **Technical Knowledge.** Ability to program, knowledge of underlying technology and applicable tools, and an understanding of the dynamics of software development generally, though not in every sense, makes testing easier for us.
- **Domain Knowledge.** The more we know about the users and their problems, the better we can test.
- **Testing Skill.** Our ability to test in general obviously makes testing easier. Relevant aspects of testing skill include experiment design, modeling, product element factoring, critical thinking, and test framing.
- **Test Strategy.** A well-designed test strategy may profoundly reduce the cost and effort of testing.
- **Supporting Tester Availability.** A "supporting tester" is anyone who is not a regular tester and who is not responsible for the testing, yet is available to help test the product. Developers can be good supporting testers.

Intrinsic Testability

- **Observability.** To test we must be able to see the product. Ideally we want a completely transparent product, where every fact about its states and behavior, including the history of those facts is readily available to us.
- **Controllability.** To test, we must be able to visit the behavior of the product. Ideally we can provide any possible input and invoke any possible state, combination of states, or sequence of states on demand, easily and immediately.
- **Algorithmic Simplicity.** To test, we must be able to visit and assess the relationships between inputs and outputs. The more complex and sensitive the behavior of the product, the more we will need to look at.
- **Explainability.** To test, we must understand the design of the product as much as we can. A product that behaves in a manner that is explainable to outsiders is going to be easier to test. "Explainability" is also a hot topic in AI.
- **Unbugginess.** Bugs slow down testing because we must stop and report them, or work around them, or in the case of blocking bugs, wait until they get fixed. It's easiest to test when there are no bugs.
- **Smallness.** The less there is of a product, the less we have to look at and the less chance of bugs due to interactions among product components.
- **Decomposability.** When different parts of a product can be separated from each other, we have an easier time focusing our testing, investigating bugs, and retesting after changes.
- **Similarity** *(to known and trusted technology).* The more a product is like other products we already know the easier it is to test it. If the product shares substantial code with a trusted product, or is based on a trusted framework, that's especially good.

Risk Analysis Heuristics *(for Digital Products)*

By James Bach and Michael Bolton v. 2.1

This is a set of guideword heuristics for use in analyzing product risks for digital products, mainly software. "Guidewords" are words or phrases that help focus your attention on potentially important factors. Guidewords are not mutually exclusive—they interact and overlap to some degree. But that's okay. In heuristic risk analysis we do not use mathematics to calculate risk, however if many of these guidewords seems to apply to a particular component of your product, you will probably consider that part more likely to harbor serious bugs, and more worth testing.

Project Factors

Things going on in projects, among people, may lead to bugs.

Learning Curve: When developers are new to a tool, technology, or solution domain, they are likely to make mistakes. Many of those mistakes they will be unable to detect.

Poor Control: Code and other artifacts may not be under sufficient scrutiny or change control, allowing mistakes to be made and to persist. Also people may try to subvert weak controls when they perceive themselves to be under time pressure.

Rushed Work: The amount of work exceeds the time available to do it comfortably. Corners are likely to be cut; details are likely to be forgotten.

Fatigue: Programmers and other members of the development team are more likely to make mistakes when they're physically tired or even just bored.

Overfamiliarity: When people are immersed in a project or a community for an extended time, they may become blind to patterns of risks or problems that are easy for an outsider to see.

Distributed Team: When people are working remotely from each other, communication may become strained and difficult, simple collaborations become expensive, the conditions for the exchange of tacit knowledge are inhibited.

Third-party Contributions: Any part of a product contributed by a third-party vendor may contain hidden features and bugs, and the developers may otherwise not fully understand it.

Bad Tools: The project team may be saddled with tools that interfere with or constrain their work; or that may introduce bugs directly into their work.

Expense of Fixes: Some components or type of bugs may be especially expensive to fix, or take a long time to fix (platform bugs are typically like this). In that case, you may need to focus on finding those bugs especially soon.

Not Yet Tested: Any part of the product that hasn't yet been tested is obviously likely to have fresh bugs in it, compared to things that *have* been tested. Therefore, for instance, it may be better to focus on parts of the product that have not been unit tested.

Technology Factors

The structure and dynamics of technology itself may give rise to bugs.

New Technology: Over time, the risks associated with any new kind of technology will become apparent, so if your product uses the latest whizzy concept, it is more likely to have important and unknown bugs in it.

New Code: The newer the code you are testing, the more likely it is to have unknown problems.

Old Code: A product that has been around for a while may contain code that is unsuited to its current context, difficult to understand, or hard to modify.

Changed Code: Any recently changed code is more likely to have unknown problems.

Brittle Code: Some code may be written in a way that makes it difficult to change without introducing new problems. Even if this code never changes, it may be brittle in the sense that it tends to break when anything around it changes.

Complexity: The more different interacting elements a product has, the more ways it can fail; the more states or state transitions it has, the more states can be wrong.

Failure History: The more that a product or part of a product has failed in the past, the more you might expect it to fail in the future. Also, if a particular product has failed in a particularly embarrassing way, it perhaps should not be allowed to fail in that way again without bring the project team into disrepute.

Dependencies Upstream: One part of a system or one feature of a product may depend on data or conditions that are controlled by other components that come before it. The more upstream processing that must occur correctly, the more likely that any bugs in those processes may cause failure in the downstream component.

Dependencies Downstream: Any particular component that has many other components that rely on it will involve more risk, because the upstream bugs will propagate trouble downstream.

Distributed Components: A product may be comprised of things that spread out over a large area, connected by tenuous network links that introduce uncertainty, noise, or lag time into the system.

Open-Ended Input: The greater freedom there is in data, the more likely that a particular configuration of data could trigger a bug. Lack of filtering and bounding are especially a problem for security.

Hard to Test: When something is hard to test, perhaps because it is hard to observe or hard to control, there will be greater risk that bugs will go undetected, and it will require extra effort to find the important bugs.

Hardware: Hardware components can't be changed easily. Hardware related problems must be found early because of the long lead time for fixing.

Requirements Factors

Aspects of requirements may indicate or promote the presence of bugs.

Ambiguity: Words and diagrams are always interpreted by people, and different people will often have different interpretations of things. More ambiguity means more likelihood that a bug can be introduced through honest misunderstanding.

Very High Precision: Sometimes a document will specify a higher level of precision than is necessary or achievable. Sometimes the product should behave in a way that is more precise than the specification suggests. In any case, higher the precision required, the more likely it is that the product will not meet that requirement.

Mysterious Silence: Sometimes a specification will leave out things that a tester might think are essential or important. This "mysterious" silence might indicate that the designers are not thinking enough about those aspects of the design, and therefore there are perhaps more bugs in it. This is commonly seen with error handling.

Undecided Requirements: The designers might have intentionally left parts of the product unspecified because they don't yet know how it should work. Postponing the design of a system is a normal part of Agile development, for instance, but wherever that happens there is a possibility that a big problem will be hiding in those unknown details.

Evolving Requirements: Requirements are not static, they are changed and developed and extended. Any document is a representation of what some person believed at some time in the past; and when when a requirement is updated, it's possible that other requirements which SHOULD have changed, didn't. Fast evolving requirements often develop inconsistencies and contradictions that lead to bugs.

Imported Requirements: Sometimes requirement statements are "borrowed"— cut and pasted from other documents or even from other projects. These may include elements not appropriate to the current project.

Hard to Read: If the document is large, poorly formatted, repetitive, or otherwise hard to read, it is less likely to have been carefully written or properly reviewed.

Non-Native Writers: When the person writing the specification is not fluent in the specification's language, misunderstanding and error are likely.

Non-Native Readers: When the people reading and interpreting the specification are not fluent in the specification's language, misinterpretation is likely.

Critical Feature: The more important a feature is, the more important its bugs will be.

Strategic Feature: A feature might be key to differentiating your product from a competitor; or might have a special notoriety that would make its bugs especially important.

VIP Opinion: A particular important person might be paying attention to a particular feature or configuration or type of use, making bugs in that area more important. Or the important person's fascination with one aspect of the product may divert needed attention from other parts of the product.

Operational Factors

The circumstances and patterns of use affect the probability and impact of bugs.

Popular Feature: The more people use a feature, the more likely any bugs in it will be found by users.

Disconnection: Different parts of a product that must work together may fall into incompatible states, leading to a failure of the system as a whole.

Unreliable Platform: Deployed products may exhibit problems due to variations or failures in the underlying supporting technology.

Security Threats: Malicious actors will attempt to break in.

Misusable: A feature might be easily misused, such that it might misbehave in a way that while not technically a flaw in the design, is still effectively a bug.

Glaring Failure: A problem or its consequences may be obvious to anyone who encounters it.

Insidious Failure: The causes or symptoms of a problem may be invisible or difficult to see for some time before they are noticed, allowing more trouble to build.

Is the Product Good Enough?

A Heuristic Framework for Thinking Clearly About Quality

GEQ Perspectives

1. **Stakeholders:** *Whose opinion about quality matters? (e.g. project team, customers, trade press, courts of law)*
2. **Mission:** *What do we have to achieve?(e.g. immediate survival, market share, customer satisfaction)*
3. **Time Frame:** *How might quality vary with time?(e.g. now, near-term, long-term, after critical events)*
4. **Alternatives:** *How does this product compare to alternatives, such as competing products, services, or solutions?*
5. **Consequences of Failure:** *What if quality is a bit worse than good enough? Do we have a contingency plan?*
6. **Ethics:** *Would our standard of quality seem unfairly or negligently low to a reasonable observer?*
7. **Quality of Assessment:** *How confident are we in our assessment? Do we know enough about this product?*

GEQ Factors

1. **Assess the benefits of the product:**
   - 1.1 **Identification:** *What are the benefits or potential benefits for stakeholders of the product?*
   - 1.2 **Likelihood:** *Assuming the product works as designed, how likely are stakeholders to realize each benefit?*
   - 1.3 **Impact:** *How desirable is each benefit to stakeholders?*
   - 1.4 **Individual Criticality:** *Which benefits, all by themselves, are indispensable?*
   - 1.5 **Overall Benefit:** *Taken as a whole, and assuming no problems, are there sufficient benefits for stakeholders?*
2. **Assess the problems of the product:**
   - 2.1 **Identification:** *What are the problems or potential problems for stakeholders of the product?*
   - 2.2 **Likelihood:** *How likely are stakeholders to experience each problem?*
   - 2.3 **Impact:** *How damaging is each problem to stakeholders? Are there workarounds?*
   - 2.4 **Individual Criticality:** *Which problems, all by themselves, are unacceptable?*
   - 2.5 **Overall Impact:** *How do all the problems add up? Are there too many non-critical problems?*
3. **Assess product quality:**
   - 3.1 **Overall Quality:** *With respect to the GEQ Perspectives, do the benefits outweigh the problems?*
   - 3.2 **Margin of Safety/Excellence:** *Do benefits to outweigh problems to a* sufficient degree *for comfort?*
4. **Assess our capability to improve the product:**
   - 4.1 **Strategies:** *Do we know how the product could be noticeably improved?*
   - 4.2 **People & Tools:** *Do we have the right people and tools to implement those strategies?*
   - 4.3 **Costs:** *How much cost or trouble will improvement entail? Is that the best use of resources?*
   - 4.4 **Schedule:** *Can we ship now and improve later? Can we achieve improvement in an acceptable time frame?*
   - 4.5 **Benefits:** *How specifically will it improve? Are there any side benefits to improving it (e.g. better morale)?*
   - 4.6 **Problems:** *How might improvement backfire (e.g. introduce bugs, hurt morale, starve other projects)?*

In the present situation, all things considered, is it more harmful than helpful to further improve the product?

About this Framework

This analysis framework represents one of many ways to reason about Good Enough quality. It's based on this assertion:

A product is good enough when *all* of these conditions apply:

1. *It has sufficient benefits.*
2. *It has no critical problems.*
3. *The benefits sufficiently outweigh the problems.*
4. *In the present situation, and all things considered, further improvement would be more harmful than helpful.*

Each point, here, is critical. If any one of them is not satisfied, then the product, although perhaps good, cannot be good *enough*. The first two seem fairly obvious, but notice that they are not exact opposites of each other. The complete absence of problems cannot guarantee infinite benefits, nor can infinite benefits guarantee the absence of problems. Benefits and problems do offset each other, but it's important to consider the product from both perspectives. Point #3 reminds us that benefits must not merely outweigh problems, they must do so to a *sufficient* degree. It also reminds us that even in the absence of any individual critical problem, there may be patterns of non-critical problems that essentially negate the benefits of the product. Finally, point #4 introduces the important matter of logistics and side effects. If high quality is too expensive to achieve, or achieving it would cause other unacceptable problems, then we either have to accept lower quality as being good enough or we have to accept that a good enough product is impossible.

The analysis framework (p. 1) is a more detailed expression of the basic Good Enough model. It is meant to jog your mind about every important aspect of the problem. To apply it, think upon each of the *GEQ Factors* in light of each of the *GEQ Perspectives*. This process can be helpful in several ways:

1. Use it to make a solid argument in favor of further improvement. For instance, you might apply the stakeholder and critical purpose perspectives to support an argument that a particular packaged software product under development, while possessing cool features that will please enthusiasts, does not possess certain benefits that mainstream customers require (e.g. convenient data interchange with Microsoft Office). Mainstream customers may also require higher reliability.

2. Use it to explore how to invest *now* to support higher standards *later*. If you know at the beginning of a project that there will be tough quality decisions to make at the end, you can work to assure that the quality bar will be set high. Looking at the framework, you can see that by lowering the cost of improvement, it may be less of a burden and can go on longer. Preventing problems could cause higher quality to be attainable in the same time frame.

3. Use it to form your own notion of acceptable quality. There's nothing sacred about this framework. It's a work in progress. Hold your idea of quality as clearly as you can in your mind's eye, then run through the framework and see if you find any of the questions jarring or unnecessary. Try to trace the source of your discomfort. Do you prefer different terminology? A model that more closely fits your technology or market? Are there any missing questions?

Why "Good Enough?"

Software quality assessment is a hard problem. Although there are many interesting measurable quality factors, there is no conceivable single measure that represents all that we mean by the word quality. Since quality is multidimensional and ultimately a subjective idea, a responsible and accurate perception of it must be constructed in our minds from all the facts and perceptions. It's a cognitive process akin to analyzing the stock market, or handicapping racehorses.

When it comes to *maximizing* software quality, we have another hard problem-- how good is good enough? Quality is not free, we have to exert ourselves to achieve it. At what point does it make more sense to turn our attention from improving a particular product to shipping that product, or at the very least, improving something else? How best can we motivate management to invest in processes and systems that lead to higher quality for less effort? We can strive for perfection, but what if we run out of time before we achieve that worthy goal? Wouldn't it be helpful to form an idea of good enough quality, just in case perfection proves itself to be out of reach? We also need to consider that "as good as we possibly can do" might not be good enough. Even perfection might not be good enough if we seek to achieve something that's impossible to begin with. No matter what we want to achieve, it sure comes in handy to consider the dynamics of required quality vs. desired quality.

Bug Fix Analysis

Problem Analysis

Frequency

1.1 How was the bug found?
- 1.1.1 Was it found by a user?
- 1.1.2 Is it a natural or contrived case?
- 1.1.3 Is it a typical or pathological case?
- 1.1.4 Was the bug caused by a recent fix to another bug?

1.2 How often is it likely to occur?
- 1.2.1 Is it intermittent or predictable?
- 1.2.2 Is it a one-time problem or ongoing?

1.3 How soon after the bug was created did we discover it?

Severity

2.1 Does the bug cause any user data to be lost?

2.2 Will it cause an additional load for Technical Support?

2.3 How likely is the user to notice it when it occurs?

2.4 Is it the tip of an iceberg?
- 2.4.1 Will it trigger other problems?
- 2.4.2 Is it part of a class of bugs that should all be fixed?
- 2.4.3 Does it represent a basic design deficiency?

2.5 Was this bug shipped in the previous release?
- 2.5.1 Did Technical Support hear anything about it?
- 2.5.2 Has anything changed since the last version that would make it more or less of a problem?

2.6 Is this bug less severe than others we've deferred? more severe than others we've fixed?

Publicity

3.1 Are certain kinds of users more likely to be affected than others?
- 3.1.1 How sophisticated are those users?
- 3.1.2 How vocal are those users?
- 3.1.3 How important are those users?
- 3.1.4 Will it affect the review writers at any major magazines?

3.2 Are our competitors strong or weak in the same functional areas?

3.3 Is this the first release of this feature or is there an installed base?

3.4 Is the problem so esoteric that no one will notice before we can update the product?

3.5 Does it look like a defect to the casual observer, or like a natural limitation?

Solution Analysis

Identification

4.1 Is the solution related to third-party components?

4.2 What are the workarounds?

- 4.2.1 Are they obvious or esoteric?

4.3 Can we "document around it" instead of fixing it?

4.4 Can the solution be postponed until the next release?

4.5 Is a fix known?

- 4.5.1 Are there several possible fixes or just one?
- 4.5.2 How many lines of code are involved?
- 4.5.3 Is it complex code or simple code?
- 4.5.4 Is it familiar code or legacy code?
- 4.5.5 Is the fix a tweak, rewrite, or substantial new code?
- 4.5.6 How long will it take to implement the fix?
- 4.5.7 What components are affected by the fix?
- 4.5.8 Will it require rebuilds of dependent components?
- 4.5.9 Does the fix impact documentation in any way? screenshots? online help?

Verification

5.1 What new problems could the fix cause? worst case?

5.2 How effectively could we test the fix, if we authorize it?

- 5.2.1.1 Was this bug found late in the project? Does that indicate a weakness in the test suite?
- 5.2.1.2 Will the test automation cover this case?
- 5.2.1.3 Could the fix be sent specially to some or all of the beta testers?

5.3 How hard would it be to undo the fix, if there's trouble with it?

Perspective

6.1 How dangerous is it to make changes in this code?

6.2 Will a fix to this component be the only reason to rebuild or remaster?

6.3 Who wants this fix internally? What are the politics involved?

6.4 How does the overall quality compare to previous releases?

6.5 If we think this bug is important, why not slip the schedule by two weeks and fix more bugs?

6.6 What would be the right thing to do? the safe thing to do?

Prevention

7.1 Was the problem caused by a fix approved after code freeze?

7.2 What was the error that caused the defect?

7.3 Is there any internal error checking or unit test that should be added to catch bugs of this type?

7.4 Is there any review process that could catch bugs like this before they get into the build?

A Concise QA Process

(Developed by me, James Bach, for a start-up market-driven product company with a small base of customers, this process is intended to be consistent with the principles of the Context-Driven School of testing and the Rapid Testing methodology. Although it is not a "best practice", I offer it as an example of how a concise QA process might look.)

This document describes the basic terminology and agreements for an agile QA process.

If these ideas don't seem agile to you, *question them*, then *change them*.

Build Protocol

Addresses the problem of wasting time in a handoff from development to testing.

- ❑ [When time is of the essence] Development alerts testing as soon as they know they'll be delivering a build.
- ❑ Development sends testing at least a bullet list describing the changes in the build.
- ❑ Development is available to testers to answer questions about fixes or new features.
- ❑ Development updates bug statuses in the bug tracking system.
- ❑ Development builds the product based on version controlled code, according to a repeatable build process, stamping each build with unique version number.
- ❑ When the build is ready, it is placed on the server.
- ❑ Testing commits to reporting sanity test status within one hour of build delivery.

Test Cycle Protocol

Addresses the problem of diffusion of testing attention and mismatch of expectations between testing and its clients.

There are several kinds of test cycle:

- ❑ *Full cycle:* All the testing required to take a releasable build about which we know nothing and qualify it for release. A full test cycle is a rare event.
- ❑ *Normal cycle:* This is either an incremental test cycle, during Feature Freeze or Code Freeze, based on testing done for earlier builds, or it's an interrupted cycle, which ends prematurely because a new build is received, or because testing is called off.
- ❑ *Spot cycle:* This is testing done prior to receiving a formal build, at the spontaneous request of the developer, to look at some specific aspect of the product.
- ❑ *Emergency cycle:* "Quick! We need to get this fix out." If necessary testing will drop everything and, without prior notice, can qualify a release in hours instead of days. This would be a "best effort" test process that involves more risk of not catching an important bug.

What happens in a test cycle:

- ❑ *Perform smoke test right away.*
- ❑ *Install product in test lab.*
- ❑ *Run convenient test automation.*
- ❑ *Verify bug fixes.*
- ❑ *Test new stuff.*
- ❑ *Re-test anything suspected to be impacted by changes.*
- ❑ *Periodically re-test things not tested recently.*
- ❑ *Periodically re-test previously fixed bugs.*
- ❑ *Perform "enabled" test activities (what recent additions or fixes make possible).*
- ❑ *Revisit mystery bugs.*
- ❑ *Continue previous test cycle.*
- ❑ *Investigate and report problems; otherwise provide quick feedback to development.*
- ❑ *Coordinate help from part-time testers.*

Change Protocol

Addresses the problem of excessive retesting or failure to detect important problems late in the development cycle.

Release Team: This is the person or persons who make the decision (or substantially contribute to the decision) to release the product. Typically includes development manager, test manager, product manager, and project manager.

There are different levels of change control because we have competing goals. We want to get the job done fast, and we want to get it done right. This calls for phased change control. Freezing allows testing to run briefer test cycles.

On any real project, some of these phases may be skipped. A small release might go directly to code freeze.

- ❑ *Alpha:* Development manages changes within itself. No externally imposed protocol.
- ❑ *Feature Freeze:* Typically begins with the delivery of a feature complete build. No new features without specific Release Team approval. Any bug fix can be made without approval.
- ❑ *Code Freeze:* Typically begins with the delivery of a release candidate. No changes of any kind can be made without specific approval by the Release Team.

The release team must meet periodically, perhaps every day, during freezes. They look over change requests and bugs and decide what will be done.

Release Protocol

Addresses the problem of messing up at the very last minute.

- *Signoff:* The release team formally decides that a particular release candidate can be shipped.
- *Package testing:* Testing performs final checks, including a virus scan, release notes review, and file version review. Final installation testing.
- *FCS:* Final customer ship.
- *Acceptance Testing:* Customer installs and tests product while testers and developers stand by to support.

Putt Putt Saves the Zoo

(Product coverage outline after 1 hour)

- Plot Line
  - pre-rescue parental conversations
  - post-rescue parental conversations
  - changing baby conversations & sound bites
  - Pre-Rescue Sequences
  - Post-Rescue Sequences
- Conversations
- Characters
  - ShopKeeper
  - Food Cart
  - Gift Cart
  - Outback Al
  - Animal Parents
  - Animal Babies
  - Putt-Putt
- Props
  - List of Animals
  - Map of Zoo
  - Zoo Chow
  - Dog
  - Rope
  - Shovel
  - Hot Cocoa
  - Toolbox
  - Log
  - Raft
  - Cheese puffs
  - Camera
- Screens
  - Screen states
  - General
  - Special
    - Seal slide
    - Rapids
    - Alligator Bridge
    - Props
      - Snapshots
      - Toolbox
      - List of animals
- Sprites
  - Stateless
  - State-Based
    - One shot
    - Random
    - Cyclic

- Words
- Gamettes
  - Tag
  - Hockey
  - Paint Shack
- Rescue Gamettes
  - Tools
  - Icebergs
  - Cocoa
  - Rope
  - Drawbridge

Table formatting Test Notes

(After 60 Minutes)

Issues

- This is a very complex feature set. There appear to be many interesting interactions.
- The analysis, below, is not complete. We need to continue to refine and enhance it.
- What is the error handling philosophy, here?
- Is there a debug version of this?
- Is there a tool that the other testers use to test this?

Process
---
* Functional analysis
    - Most of what I did was preparatory to creating an inventory of test requirements.

* Functional exploration
    - briefly reviewed help
    - toured the menus and functions of Word that were related to table formatting.
    - contrived new table data and reviewed some existing Word files.
    - applied various stressing strategies (not systematically)
    - I did *not* apply a very precise oracle for most of what I did.

Strategy ideas

Stress test (contrived data and natural data)
Buffer overflow attack
Edit a large book.
Convert a WordPerfect file and work with tables in it.
Convert a web page from HTML and work with a table in it.
Review existing bug reports, or talk to a support guy.
Pairs matrix?
Use a table generation tool

Functions

Table Menu
    insert
    select
    delete
    convert
    Autoformatting
    Drawing Tables

Context Menu
    table properties
    (more)

Elements of Tables
    Cells
    Cells across
    Cells down
    interaction between cells and page breaks
    Long tables
    repeat headings (page breaks)

```
Elements of Cells
       Borders and Shading

       Fill color
       patterns
       text position
       text orientation
       text alignment
       contents
              text
              pictures
              OLE objects
              other tables

Other interesting elements
---------------------------
Document types
sequences of actions
interaction with other functions
- save as
- save and restore (format preserved?)
- spell check
- undo
- redo
- printing (compare printed with screen output)


Platform
-----------------------------
Memory
processor speed
Operating system
Accessability options
       high contrast
```

DiskMapper Test Notes

(After 30 Minutes)

FUNCTIONS

Map Drive
    ???when is drive mapped?
Drive Selection
Print Map
File Operations
    delete
    unzip/zip
    print
    run
    information
Invoke Explorer
Exit/Startup
Mapping Method
    Color Scheme
    level colors
    Color by
        levels
        age
        extension
        archive
        protected
        never used
Goto Root
Zoom in/out
Show one/many levels
General Options
Font Options
Online Help
About Box
Toolbar/Menus
Window management
Map display
    correctness of proportions
    filenames
    box graphics
    colors
    box vanish
    Status bar display
Map Behavior
    zooming
    highlighting
    updating
Settings preservation (dm32.ini)

DiskMapper Test Notes

(after 60 minutes)

The purpose of DM appears to be to provide a view of disk contents in a manner proportional to the size of each file and folder, and to support basic file operations on those contents. The proportional display is the central feature of the product.

Risks:

- disk corruption (causing/scanning)
- accidental deletion
- incorrect proportions
- files not displayed that should be
- ???spurious files displayed
- obsolete view of map
- Multi-tasking interference
- misleading coloring
- Big disks not displayed correctly
- display method corruption (accidentally messing up the settings and not being able to reset them)
- bad file information
- unreadable map printout
- system incompatibility
- poor performance
- ???crashing
- ???interference with other running apps

Major risks:

- display is substantially wrong
- file loss or corruption
- frequent crashes
- system incompatibility
- fails on large data sets

Functional areas to test:

Navigation
Mapping methods
Proportional display
File operations
Documentation
Windows compatibility
General UI

Platform:

Windows 98
2.1 gb disk drive
???bigger drive availability?
???Floppy disks?
???Servers

Test data:

???automatic generation of file structure?

files
- large (limits???)
- small (0)
- old
- new
- extension
- archive
- protection
- usage (never/not never)
- names

file groups
- large/small juxtaposed
- large number of small files

folders
- names
- deep nesting (max???)
  - overflow the colors
- ???is the root special?

Ini file settings
- valid
- randomized???

An Exploratory Tester's Notebook

Michael Bolton, DevelopSense
mb@developsense.com

Biography

Michael Bolton is the co-author (with senior author James Bach) of Rapid Software Testing, a course that presents a methodology and mindset for testing software expertly in uncertain conditions and under extreme time pressure.

A testing trainer and consultant, Michael has over 17 years of experience in the computer industry testing, developing, managing, and writing about software. He is the founder of DevelopSense, a Toronto-based consultancy. He was with Quarterdeck Corporation for eight years, during which he delivered the company's flagship products and directed project and testing teams both in-house and around the world.

Michael has been teaching software testing around the world for eight years. He was an invited participant at the 2003, 2005, 2006, and 2007 Workshops on Teaching Software Testing in Melbourne and Palm Bay, Florida; was a member of the first Exploratory Testing Research Summit in 2006. He is also the Program Chair for TASSQ, the Toronto Association of System and Software Quality, and a co-founder of the Toronto Workshops on Software Testing. He has a regular column in Better Software Magazine, writes for Quality Software (the magazine published by TASSQ), and sporadically produces his own newsletter.

Michael lives in Toronto, Canada, with his wife and two children.

Michael can be reached at mb@developsense.com, or through his Web site, http://www.developsense.com

Abstract: One of the perceived obstacles towards testing using an exploratory testing approach is that exploration is unstructured, unrepeatable, and unaccountable, but a look at history demonstrates that this is clearly not the case. Explorers and investigators throughout history have made plans, kept records, written log books, and drawn maps, and have used these techniques to record information so that they could report to their sponsors and to the world at large. Skilled exploratory testers use similar approaches to describe observations, to record progress, to capture new test ideas, and to relate the testing story and the product story to the project community. By focusing on what actually happens, rather than what we hope will happen, exploratory testing records can tell us even more about the product than traditional pre-scripted approaches do.

In this presentation, Michael Bolton invites you on a tour of his exploratory testing notebook and demonstrates more formal approaches to documenting exploratory testing. The tour includes a look at an informal exploratory testing session, simple mapping and diagramming techniques, and a look at a Session-Based Test Management session sheet. These techniques can help exploratory testers to demonstrate that testing has been performed diligently, thoroughly, and accountably in a way that gets to the heart of what excellent testing is all about: a skilled technical investigation of a product, on behalf of stakeholders, to reveal quality-related information of the kind that they seek.

Documentation Problems

There are many common claims about test documentation: that it's required for new testers or share testing with other testers; that it's needed to deflect legal liability or to keep regulators happy; that it's needed for repeatability, or for accountability; that it forces you to think about test strategy. These claims are typically used to support heavyweight and formalized approaches to test documentation (and to testing itself), but no matter what the motivation, the claims have this in common: they rarely take context, cost, and value into account. Moreover, they often leave out important elements of the story. Novices in any discipline learn not only through documents, but also by observation, participation, practice, coaching, and mentoring; tester may exchange information through conversation, email, and socialization. Lawyers will point out that documentation is only one form of evidence—and that evidence can be used to buttress or to skewer your case—while regulators (for example, the FDA[1]) endorse the principle of the least burdensome approach. Processes can be repeatable without being documented (how do people get to work in the morning?), and auditors are often more interested in the overview of the story than each and every tiny detail. Finally, no document—least of all a template—ever *forced* anyone to think about anything; the thinking part is always up to the reader, never to the document.

Test documentation is often driven by templates in a way that standardizes look and feel without considering content or context. Those who set up the templates may not understand testing outside the context for which the template is set up (or they may not understand testing at all); meanwhile, testers who are required to follow the templates don't own the format. Templates—from the IEEE 829 specification to Fitnesse tests on Agile projects—can standardize and formalize test documentation, but they can also standardize and formalize thinking about testing and our approaches to it. Scripts stand the risk of reducing learning rather than adding to it, because they so frequently leave out the motivation for the test, alternative ways of accomplishing the user's task, and variations that might expose bugs.

Cem Kaner, who coined the term exploratory testing in 1983, has since defined it as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, and execution as mutually supportive activities that run in parallel throughout the project."[2] A useful summary is "simultaneous test design, test execution, and learning." In exploratory testing, the result of the last test strongly influences the tester's choices for the next test. This suggests that exploratory testing is incompatible with most formalized approaches to test documentation, since most of them segregate design, execution, and learning; most emphasize scripted actions; and most try to downplay the freedom and responsibility of the individual tester. Faced with this problem, the solution that many people have used is simply to avoid exploratory testing—or at least to avoid admitting that they do it, or to avoid talking about it in reasonable ways. As

---

[1] *The Least Burdensome Provisions of the FDA Modernization Act of 1997; Concept and Principles; Final Guidance for FDA and Industry.* www.fda.gov/cdrh/modact/leastburdensome.html

[2] This definition was arrived at through work done at the 2006 Workshop on Heuristic and Exploratory Testing, which included James Bach, Jonathan Bach, Scott Barber, Michael Bolton, Tim Coulter, Rebecca Fiedler, David Gilbert, Marianne Guntow, James Lyndsay, Robert Sabourin, and Adam White. The definition was used at the November 2006 QAI Conference. Kaner, "Exploratory Testing After 23 Years", www.kaner.com/pdfs/ETat23.pdf

McLuhan said, "We shape our tools; thereafter our tools shape us."[3] Test documentation is a tool that shapes our testing.

Yet exploration is essential to the investigative dimension of software testing. Testing that merely confirms expected behaviour can be expected to suffer from fundamental attribution error ("it works"), confirmation bias ("all the tests pass, so it works"), and anchoring bias ("I *know* it works because all the tests pass, so it works"). Testers who don't explore the software fail to find the bugs that real users find when *they* explore the software. Since any given bug is a surprise, no script is available to tell you how to investigate that bug.

Sometimes documentation is a product, a deliverable of the mission of testing, designed to be produced for and presented to someone else. Sometimes documentation is a tool, something to help keep yourself (or your team) organized, something to help with recollection, but not intended to be presented to anyone[4]. In the former case, presentation and formatting are important; in the latter case, they're much less important. In this paper, I'll introduce (or for some people, revisit) two forms of documentation—one primarily a tool, and the other a product—to support exploratory approaches. The first tends to emphasize the learning dimension, the latter tends to be more applicable to test design and test execution.

This paper and the accompanying presentation represent a highly subjective and personal experience report. While I may offer some things that I've found helpful, this is not intended to be prescriptive, or to offer "best practices"; the whole point of notebooks—for testers, at least—is that they become what you make of them.

An Exploratory Tester's Notebook

Like most of us, I've kept written records, mostly for school or for work, all my life. Among other means of preserving information, I've used scribblers, foolscap paper, legal pads, reporter or steno notepads, pocket notepads, ASCII text files, Word documents, spreadsheets, and probably others.

In 2005, I met Jon Bach for the first time. Jon, brother of James Bach, is an expert exploratory tester (which apparently runs in the family) and a wonderful writer on the subject of E.T., and in particular how to make it accountable. The first thing that I noticed on meeting Jon is that he's an assiduous note-taker—he studied journalism at university—and over the last year, he has inspired me to improve my note-taking processes.

The Moleskine Notebook

One factor in my personal improvement in note-taking was James Bach's recommendation of the Moleskine pocket notebook. I got my first one at the beginning of 2006, and I've been using it ever since. There are several form factors available, with soft or hard covers. The version I have fits in a pocket; it's perfect-bound so it lies flat; it has a fabric bookmark and an elasticized loop that holds the book closed. The pages can be unlined, lined, or squared (graph paper)[5]. I prefer the graph paper; I find that it helps with sketching and with laying out tables of information.

---

[3] Marshall McLuhan, *Understanding Media: The Extensions of Man (Critical Edition).* Gingko Press, Costa Madera, CA, September 2003.

[4] See Kaner, Cem; Bach, James, and Pettichord, Bret, *Lessons Learned in Software Testing*. John Wiley & Sons, New York, 2002.

[5] They can also be lined with five-line staff paper for musicians.

The Moleskine has a certain kind of chic/geek/boutique/mystique kind of appeal; it turns out that there's something of a cult around them, no doubt influenced by their marketing. Each notebook comes with a page of history in several languages, which adds to the European cachet. The page includes the claim that the Moleskine was used by Bruce Chatwin, Pablo Picasso, Ernest Hemingway, Henri Mattisse, Andre Breton, and others who are reputed to have used the Moleskine. The claim is fictitious[6], although these artists did use books of the same colour, form factor, with sewn bindings and other features that the new books reproduce. The appeal, for me, is that the books are well-constructed, beautiful, and inviting. This reminds me of Cem Kaner's advice to his students: "Use a good pen. Lawyers and others who do lots of handwriting buy expensive fountain pens for a reason. The pen glides across the page, requiring minimal pressure to leave ink."[7] A good tool asks to be used.

Why Use Notebooks?

In the age of the personal digital assistant (I have one), the laptop computer, (I have one), and the desktop computer (I have one), and the smart phone (I don't have one), why use notebooks?

- They're portable, and thus easy to have consistently available.
- They never crash.
- They never forget to auto-save.
- The batteries don't wear out, they don't have to be recharged—and they're never AA when you need AAA or AAA when you need AA.
- You don't have to turn them off with your other portable electronic devices when the plane is taking off or landing.

Most importantly, notebooks are free-form and personal in ways that the "personal" computer cannot be. Notebooks afford diversity of approaches, sketching and drawing, different thinking styles, different note-taking styles. All Windows text editors, irrespective of their features, still look like Windows programs at some level. In a notebook, there's little to no reformatting; "undo" consists of crossing out a line or a page and starting over or, perhaps more appropriately, of tolerating imperfection. When it's a paper notebook, and it's your own, there's a little less pressure to make things look good. For me, this allows for a more free flow of ideas.

Figure 1: Page from Michael Bolton's Notebook #2

In 2005, James and Jonathan Bach presented a paper at the STAR West conference on exploratory

[6] http://www.iht.com/articles/2004/10/16/mmole_ed3_.php
[7] http://www.testingeducation.org/BBST/exams/NotesForStudents.htm

dynamics, skills and tactics. Michael Kelly led a session in which we further developed this list at Consultants' Camp 2006.

Several of the points in this list—especially modeling, questioning, chartering, observing, generating and elaborating, abandoning and recovering, conjecturing, and of course recording and reporting—can be aided by the kinds of things that we do in notebooks: writing, sketching, listing, speculating, brainstorming, and journaling. Much of what we think of as history or scientific discovery was first recorded in notebooks. We see a pattern of writing and keeping notes in situations and disciplines where learning and discovery are involved. A variety of models helps us to appreciate a problem (and potentially its solution) from more angles. Thinking about a problem is different from uttering it, which is still different from sketching it or writing prose about it. The direct interaction with the ink and the paper gives us a tactile mode to supplement the visual, and the fact that handwriting is, for many people, slower than typing, may slow down our thought processes in beneficial ways. A notebook gives us a medium in which to record, re-model, and reflect. These are, in my view, essential testing skills and tactics.

From a historical perspective, we are aware that Leonardo was a great thinker because he left notebooks, but it's also reasonable to consider that Leonardo may have been a great thinker at least in part because he *used* notebooks.

Who Uses Notebooks?

Inventors, scientists, explorers, artists, writers, and students have made notebook work part of their creative process, leaving both themselves and us with records of their thought processes.

Leonardo da Vinci's notebooks are among the most famous books in history, and also at this writing the most expensive; one of them, the Codex Leicester, was purchased in 1994 for $30.8 million by a certain ex-programmer from the Pacific Northwest[8]. Leonardo left approximately 13,000 pages of daily notes and drawings. I was lucky enough to see one recently—the Codex Foster, from the collection of the Victoria and Albert Museum.

[8] Incidentally, the exhibit notes and catalog suggested that Leonardo didn't intend to encrypt his work via the mirror writing for which he was so famous; he wrote backwards because he was left-handed, and writing normally would smudge the ink.

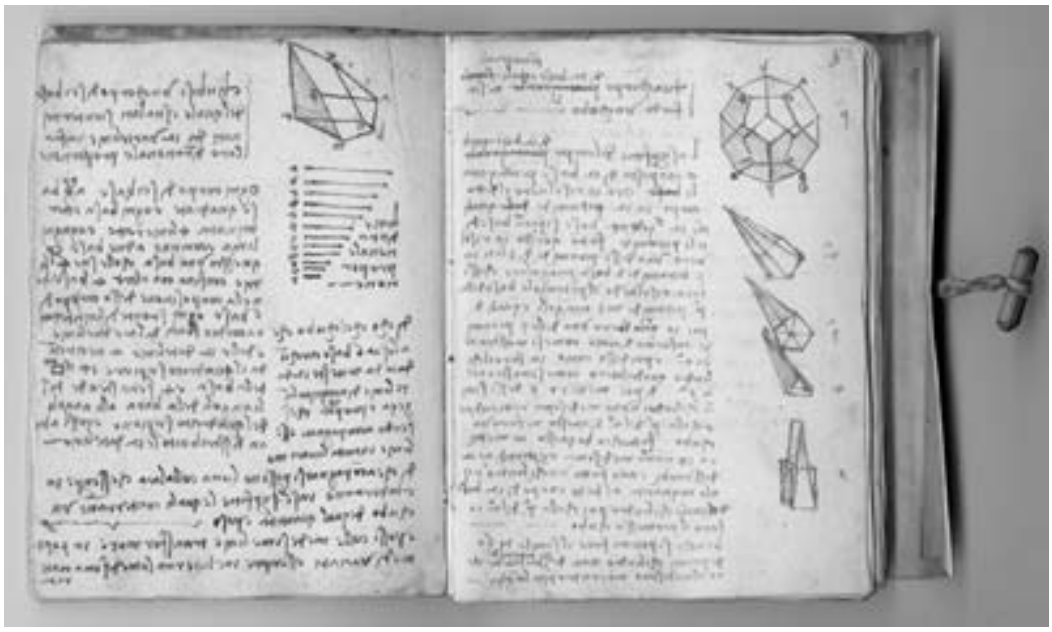


Figure 2: Leonardo da Vinci, The Codex Foster

As a man of the Renaissance, Leonardo blurred the lines between artist, scientist, engineer, and inventor[9], and his notebooks reflect this. Leonardo collects ideas and drawings, but also puzzles, aphorisms, plans, observations. They are enormously eclectic, reflecting an exploratory outlook on the world. As such, his notebooks are surprisingly similar to the notebook patterns of exploratory testers described below, though none has consciously followed Leonardo's paradigms or principles, so far as I know. The form factor is also startlingly similar to the smaller Moleskine notebooks. Obviously, the significance of our work pales next to Leonardo's, but is there some intrinsic relationship between exploratory thinking and the notebook as a medium?

What Do I Use My Notebook For?

I've been keeping three separate notebooks. My large-format book contains notes that I take during sessions at conferences and workshops. It tends to be tidier and better-organized. My small-format book is a ready place to record pretty much anything that I find interesting. Here are some examples:

Lists of things, as brainstorms or catalogs. My current lists include testing heuristics; reifications; and test ideas. These lists are accessible and can be added to or referenced at any time. This is my favorite use of the Moleskine—as a portable thinking and storage tool.

"Fieldstones" and blog entries. Collections of observations; the odd rant; memorable quotes; aphorisms. The term "fieldstone" is taken from Gerald M. Weinberg's book Weinberg on Writing: The Fieldstone Method. In the book, Jerry uses the metaphor of the pile of stones that are pulled from the field as you clear it; then you assemble a wall or a building from the fieldstones.[10] I collect ideas for articles and blog entries and develop them later.

[9] How To Think Like Leonardo da Vinci
[10] Weinberg, Gerald M., *Weinberg on Writing: The Fieldstone Method.*

Logs of testing sessions. These are often impromptu, used primarily to practice testing and reporting, to reflect and learn later, and to teach the process. A couple of examples follow below.

Meeting notes. He who controls the minutes controls history, and he who controls history controls the world.

Ultra-Portable PowerPoints. These are one-page presentations that typically involve a table or a diagram. This is handy for the cases in which I'd like to make a point to a colleague or client. Since the listener focuses on the data and on my story, and not on what Edward Tufte[11] calls "chartjunk", the portable PowerPoints may be more compelling than the real thing.

Mind maps and diagrams. I use these for planning and visualization purposes. I need to practice them more. I did use a mind map to prepare this presentation.

Notes collected as I'm teaching. When a student does something clever during a testing exercise, I don't want to interrupt the flow, but I do want to keep track of it so that I can recount it to the class and give recognition and appreciation to the person who did it. Moreover, about half the time this results in some improvement to our course materials[12], so a notebook entry is very handy.

Action items, reminders, and random notes. Sometimes the notebook is the handiest piece of paper around, so I scribble something down on a free page—contact names (for entry later), reminders to send something to someone; shopping lists.

Stuff in the pocket. I keep receipts and business cards (so I don't lose them). I also have a magic trick that I use as a testing exercise that fits perfectly into the pocket.

I try to remember to put a title and date on each page. Lately I've been slipping somewhat, especially on the random notes pages.

I've been using a second large-format notebook for notes on books that I'm studying. I haven't kept this up so well. It's better organized than my small format book, but my small format book is handy more often, so notes about books—and quotes from them—tend to go in that.

I'm not doing journaling, but the notebooks seem to remind me that, some day, I will. Our society doesn't seem to have the same diary tradition as it used to; web logs retrieve this idea. Several of my colleagues do keep personal journals.

How Do Other Exploratory Testers Use Notebooks?

I've done a very informal and decidedly unscientific survey of some of my colleagues, especially those who are exploratory testers.

---

[11] Tufte, Edward, *Envisioning Information*. Graphics Press, Chesire, Connecticut, 1990.
[12] Bach, James, and Bolton, Michael, *Rapid Software Testing*. http://www.satisfice.com/rst.pdf.

Adam White reports, “My notebook is my *life*. It's how I keep track of things I have to do. It supplements my memory so that I don't waste brain power on remembering to remember something. I just record it and move on.

“I have found a method of taking notes that brings my attention to things. If someone tells me about a book then I will write "Book" and underline it twice. Then when flipping back through my notes I can see that I have a reference to a book that I thought was interesting at some point in time. I use this process for other things like blogs, websites, key ideas, quotes etc. It makes organizing information after the fact very easy.”

Adam reports similar experiences to my own in how he came to use Moleskines. He too observed Jon Bach and James Bach using Moleskine notebooks; he too uses a selection of books—one large-form for work, one large-form for personal journaling, and a small one for portability and availability. He also says that the elastic helps to prevent him from losing pens.

Jonathan Kohl also reports that he uses notebooks constantly. “My favorite is my Moleskine, but I also use other things for taking notes. With my Moleskine, I capture test ideas; article ideas; diagrams or models I am working on for articles; teaching materials, or some other reason for an explanation to others; and testing notes[13]. I have a couple of notes to help focus me, and the rest are ideas, impressions, and the starred items are bugs. I translated the bugs into bug reports in a fault tracking system, and the other notes into a document on risk areas. For client work, I don't usually use my Moleskine for testing, since they may want my notes.” This is an important point for contractors and full-time employees; your notebook may be considered a work product—and therefore the property of your company—if you use it at work, or for work.

“I also use index cards (preferably post-it note index cards), primarily for bug reports,” continues Jonathan. “My test area is often full of post-its, each a bug, at the end of a morning or afternoon testing session. Over time, I arrange the post-its according to groups, and log them into a bug tracker or on story cards (if doing XP.) When I am doing test automation/test toolsmith work, I use story cards for features or other tasks, and others for bugs.”

Jonathan also uses graph-paper pads for notes that he doesn't need to keep. They contain rough session and testing notes; diagrams, scrawls, models, or things that he is trying to understand better; analysis notes, interview points, and anything else he’s interested in capturing. “These notes are illegible to most people other than me, and I summarize them and put what is needed into something more permanent.” This is also an important point about documentation in general: sometimes documentation is a product—a deliverable, or something that you show to or share with someone else. At other times, documentation is a tool—a personal aid to memory or thought processes.

“I worked with engineers a lot starting out, so I have a black notebook that I use to record my time and tasks each day. I started doing this as an employee, and do it as a consultant now as well.”

Fiona Charles also keeps a project-specific notebook. She uses a large form factor, so that it can accommodate 8½ x11 pages pasted into it. She also pastes a plastic pocket, a calendar, and loose notes from pre-kickoff meetings—she says that a glue stick is an essential part of the kit. In the

[13]Jonathan provides an example at http://www.kohl.ca/articles/ExploratoryTesting_MusicofInvestigation.pdf

notebook, she records conversations with clients and others in the project community. She uses clear termination line for dates, sets of notes, and "think pages."

Jerry Weinberg also uses project notebooks. On the first page, he places his name, his contact information, and offer of a reward for the safe return of the book. On the facing page, he keeps a list of contact info for important people to the project. On the subsequent pages, he keeps a daily log from the front of the book forwards. He keeps a separate list of learnings from the back of the book backward, until the two sections collide somewhere in the middle; then he starts a new book. "I always date the learnings," he says. "In fact, I date everything. You never know when this will be useful data." Like me, he never tears a page out.

Jerry is also a strong advocate of journaling[14]. For one thing, he treats starting journaling—and the reader's reaction to it—as an exercise in learning about effecting change in ourselves and in other people. "One great advantage of the journal method," he says, "is that unlike a book or a lecture, everything in it is relevant to *you*. Because each person's learning is personal, I can't you what you'll learn, but I can guarantee that you'll learn *something*." That's been my experience; the notebook reflects me and what I'm learning. It's also interesting to ask myself about the things, or kinds of things, that I *haven't* put it.

Jon Bach reports that he uses his notebooks in several modes. "'Log file', to capture the flow of my testing; 'epiphany trap', to capture "a ha!" moments (denoted by a star with a circle around it); diagrams and models—for example, the squiggle diagram when James and I first roughed out Session-Based Test Management; to-do lists—lots and of lots them, which eventually get put into Microsoft Outlook's Task Manager with a date and deadline—reminders, flight, hotel, taxi info when traveling, and phone numbers; quotes from colleagues, book references, URLs; blog ideas, brainstorms, ideas for classes, abstracts for new talks I want to do; heuristics, mnemonics; puzzles and their solutions (like on a math exam that says "show your work"); personal journal entries (especially on a plane); letters to my wife and child -- to clear my head after some heinous testing problem I might need a break from."

Jon also identifies as significant the paradigm "'NTSB Investigator.' I'll look back on my old notes for lost items to rescue—things that are may have become more important than when I first captured them because of emergent context. You would never crack open the black box of an airplane after a successful flight, but what if there was a systemic pattern of silent failures just waiting to culminate in a HUGE failure? *Then* you might look at data for a successful flight and be on the lookout for pathologies."

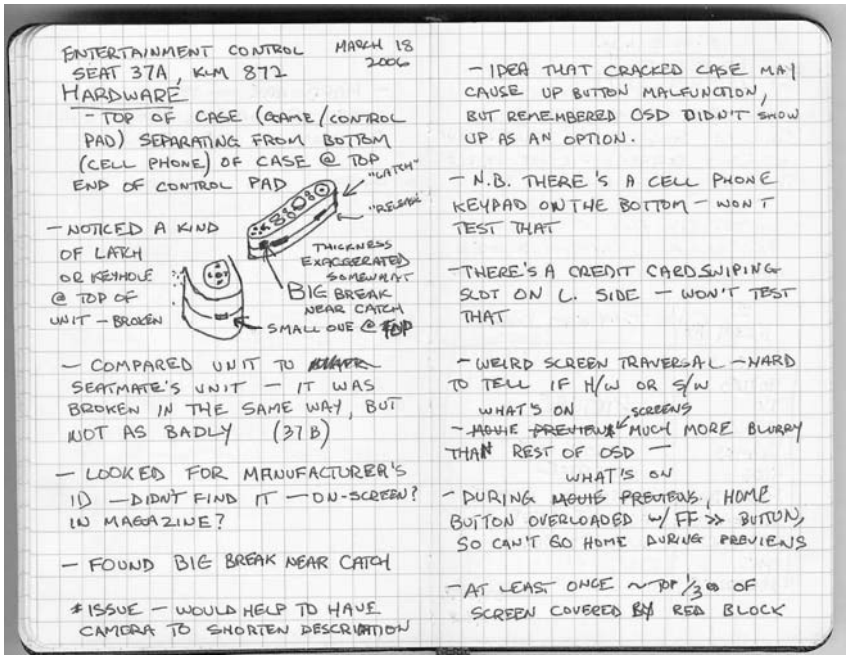
Example: An Impromptu Exploratory Testing Session

I flew from Delhi to Amsterdam. I was delighted to see that the plane was equipped with a personal in-flight entertainment system, which meant that I could choose my own movies or TV to watch. As it happened, I got other entertainment from the system that I wouldn't have predicted.

The system was menu-driven. I went to the page that listed the movies that were available, and after scrolling around a bit, I found that the "Up" button on the controller didn't work. I then inspected the controller unit, and found that it was cracked in a couple of places. Both of the

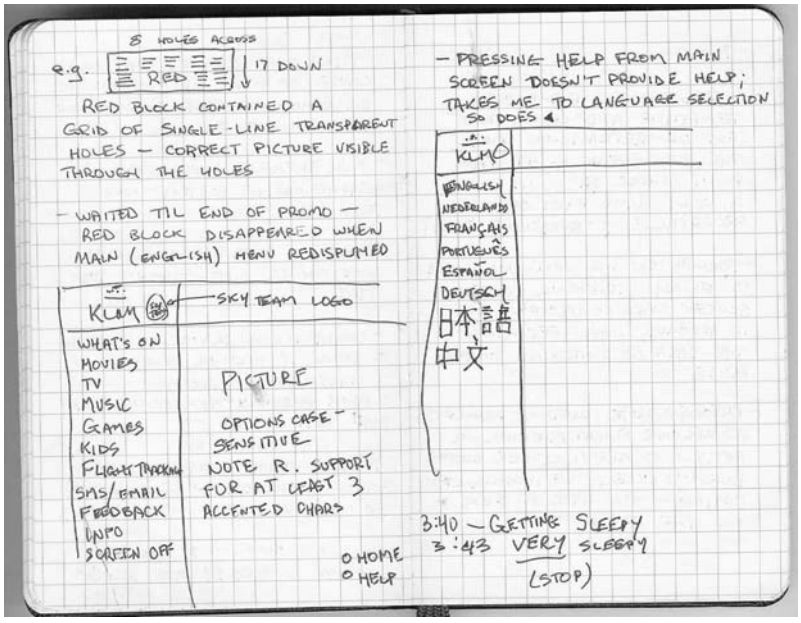
[14] Becoming a Technical Leader, pp. 80-85

cracks were associated with the mechanism that returned the unit, via a retractable cord, to a receptacle in the side of the seat. I found that if I held the controller just so, then I could get around the hardware—but the software failed me. That is, I found lots of bugs. I realized that this was an opportunity to collect, exercise, and demonstrate the sorts of note-taking that I might perform when I'm testing a product for the first time. Here are the entries from my Moleskine, and some notes about my notes.



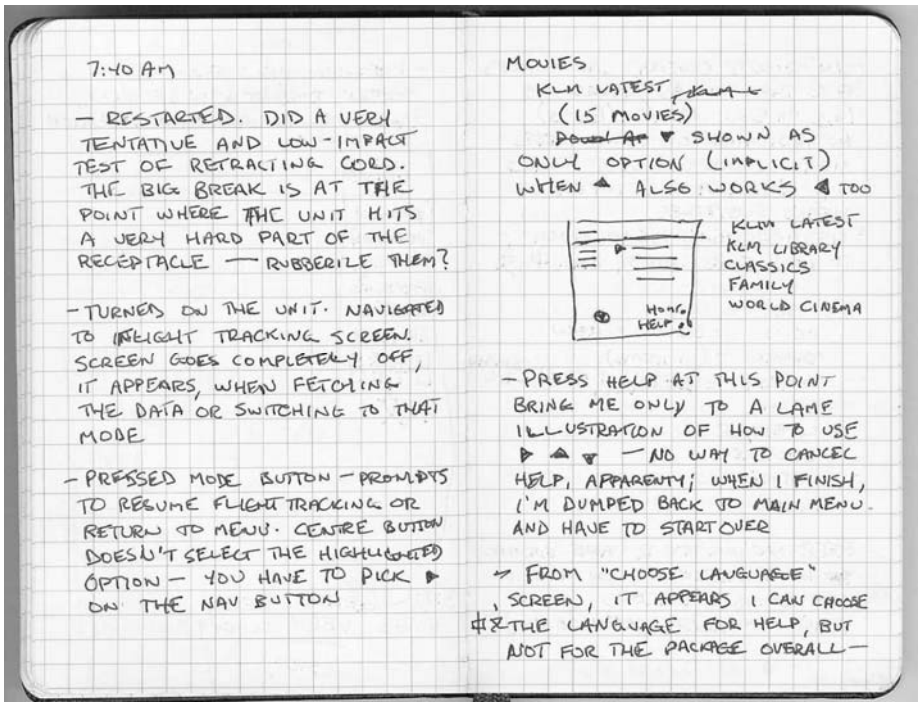
When I take notes like this, they're a tool, not a product. I don't expect to show them to anyone else; it's a possibility, but the principal purposes are to allow me to remember what I did and what I found, and to guide a discussion about it with someone who's interested.

I don't draw well, but I'm slowly getting better at sketching with some practice. I find that I can sketch better when I'm willing to tolerate mistakes.



In the description of the red block, at the top of the left page, I failed to mention that this red block appeared when I went right to the "What's On" section after starting the system. It didn't reproduce.

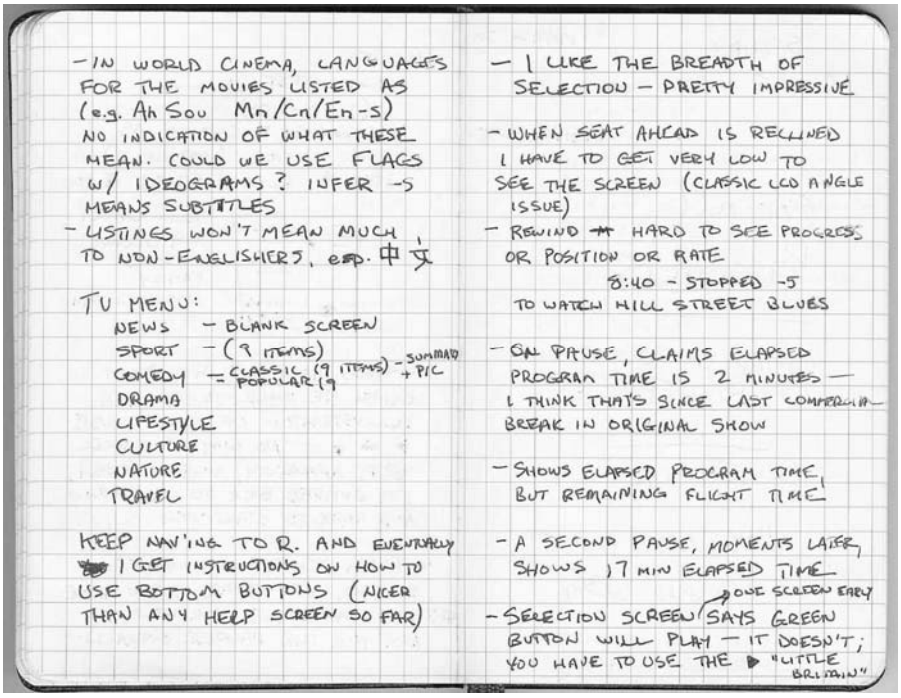
Whenever I look back on my notes, I recognize things that I missed. If they're important, I write them down as soon as I realize it. If they're not important, I don't bother. I don't feel bad about it either way; I try always to get better at it, but testers aren't omniscient. Note "getting sleepy"—if I keep notes on my own mental or emotional state, they might suggest areas that I should revisit later. One example here: on the first page of these notes, I mentioned that I couldn't find a way to contact the maker of the entertainment system. I should have recognized the "Feedback" and "Info" menu items, but I didn't; I noticed them afterwards.



After a few hours of rest, I woke up and started testing again.

Jon Bach recently pointed out to me that, in early exploration, it's often better to start not by looking for bugs, but rather by trying to build a model of the item under test. That suggests looking for the positives in the product, and following the happy path. I find that it's easy for me to fall into the trap of finding and reporting bugs. These notes reflect that I *did* fall into the trap, but I also tried to check in and return to modeling from time to time. At the end of this very informal and completely freestyle session, I had gone a long way towards developing my model and identifying various testing issues. In addition, I had found many irritating bugs.

Why perform and record testing like this? The session and these notes, combined with a discussion with the project owner, might be used as the first iteration in the process of determining an overall (and perhaps more formal) strategy for testing this product. The notes have also been a useful basis for my own introspection and critique



of my performance, and to show others some of my though process through an exploratory testing session.

A More Formal Structure for Exploratory Testing

Police forces all over the world use notebooks of some description, typically in a way that is considerably more formalized. This is important, since police notebooks will be used as evidence in court cases. For this reason, police are trained and required to keep their notebooks using elements of a more formal structure, including time of day; exact or nearest-to location; the offence or occurrence observed; the names and addresses of offenders, victims or witnesses; action taken by the officer involved (e.g. arrests), and details of conversations and other observations. (The object of the exercise here is not to turn testers into police, but to take useful insights from the process of more formal note-taking.)

How can we help to make testing similarly accountable? Session-Based Test Management (SBTM), invented by James and Jonathan Bach in 2000 is one possible answer. SBTM has as its hallmark four elements:

- Charter
- Time Box
- Reviewable Result
- Debriefing

The *charter* is a one- to three-sentence mission for a testing session. The charter is designed to be open-ended and inclusive, prompting the tester to explore the application and affording opportunities for variation. Charters are not meant to be comprehensive descriptions of what should be done, but the total set of charters for the entire project should include everything that is reasonably testable.

The *time box* is some period of time between 45 minutes and 2 ¼ hours, where a short session is one hour (+/- 15 minutes), a long session is two, and a normal session is 90 minutes. The intention here is to make the session short enough for accurate reporting, changes in plans (such as a session being impossible due to a broken build, or a session changing its charter because of a new priority), but long enough to perform appropriate setup, to get some good testing in, and to make debriefing efficient. Excessive precision in timing is discouraged; anything to the nearest five or ten minutes will do. If your managers, clients, or auditors are supervising you more closely than this,

The reviewable result takes the form of a *session sheet*, a page of text (typically ASCII) that follows a formal structure. This structure includes:

- Charter
- Coverage areas (not code coverage; typically product areas, product elements, quality criteria, or test techniques)
- Start Time
- Tester Name(s)
- Time Breakdown
    - session duration (long, normal, or short)

    - test design and execution (as a percentage of the total on-charter time)
    - bug investigation and reporting (as a percentage of the total on-charter time)
    - session setup (as a percentage of the total on-charter time)
    - charter/opportunity (expressed as a percentage of the total session, where opportunity time does not fit under the current charter, but is nonetheless useful testing work)
- Data Files
- Test Notes
- Bugs (where a "bug" is a problem that the tester and the test manager reasonably believe represents a threat to the value of the product)
- Issues (where an "issue" is a problem that threatens the value of the testing process—missing information, tools that are unavailable, expertise that might be required, questions that the tester might develop through the course of the session)

There are two reasons for this structure. The first is simply to provide a sense of order and completeness for the report and the debrief. The second is to allow a scripting tool to parse tagged information from the session sheets, such that the information can be sent to other applications for bug reporting, coverage information, and inquiry-oriented metrics gathering. The SBTM package, available at http://www.satisfice.com/sbtm, features a prototype set of batch files and Perl scripts to perform these tasks, with output going to tables and charts in an Excel spreadsheet.

The *debrief* is a conversation between the tester[15] who performed the session and someone else—ideally a test lead or a test manager, but perhaps simply another tester. In the debrief, the session sheet is checked to make sure that it's readable and understandable; the manager and the tester discuss the bugs and issues that were found; the manager makes sure that the protocol is being followed; and coaching, mentoring, and collaboration happen. A typical debrief will last between five to ten minutes, but several things may add to the length. Incomplete or poorly-written session sheets produced by testers new to the approach will prompt more questions until the tester learns the protocol. A highly complex or risky product area, a large number of bugs or issues, or an unfamiliar product may also lead to longer conversations.

Several organizations have reported that scheduling time for debriefings is difficult when there are more than three or four testers reporting to the test manager or test lead, or when the test manager has other responsibilities. In such cases, it may be possible to have the testers debrief each other.

At one organization where I did some consulting work, the test manager was also responsible for requirements development and business analysis, and so was frequently unavailable for debriefings. The team chose to use a round-robin testing and debriefing system. For a given charter, Tester A performed the session, Tester B debriefed Tester A, and at regression testing time, Tester C took a handful of sheets and used them as a point of departure for designing and executing tests. For the next charter, Tester B performed the testing, Tester C the debrief, and Tester A the regression; and so forth. Using this system, each tester learned about the product and shared information with others by a variety of means—interaction with the product, conversation in the debrief, and written session sheets. The entire team reported summaries of

[15] Or "testers"; SBTM can be used with paired testers.

the debriefings to the test manager when he was not available, and simply debriefed directly with him when he was.

Two example session sheets follow. The first is an account of an early phase of exploratory testing, in which the testers have been given the charter to create a test coverage outline and a risk list. These artifacts themselves can be very useful, lightweight documents that help to guide and assess test strategy. Here the emphasis is on learning about the product, rather than searching for bugs.

The second is an account of a later stage of testing, in which the tester has sufficient knowledge about the product to perform a more targeted investigation. In this session, he finds and reports several bugs and issues. He identifies moments at which he had new test ideas and the motivations for following the lines of investigation.

Example: Session Sheet for a Reconnaissance Session


```
CHARTER
-----------------------------------------------
Create a test coverage outline and risk list for DecideRight.

#AREAS
DecideRight
OS | Win98
Build | 1.2
Strategy | Exploration & Analysis

START
-----------------------------------------------
4/16/01 1:00pm

TESTER
-----------------------------------------------
Jonathan Bach
Tim Parkman

TASK BREAKDOWN
-----------------------------------------------

#DURATION
short

#TEST DESIGN AND EXECUTION
100

#BUG INVESTIGATION AND REPORTING
0

#SESSION SETUP
0

#CHARTER VS. OPPORTUNITY
100/0

DATA FILES
-----------------------------------------------
tco-jsb-010416-a.txt
rl-jsb-010416-a.txt

TEST NOTES
-----------------------------------------------
Tim and I walked through the User Guide table of contents and index to create
the following TCO:

Operating Systems:

      Win98
      Win2000

General Features:

      Installation
      User Manual
      Online Help
      UI
      Preferences
```

Test coverage is not merely code coverage. Functional areas, platforms, data, operations, and test techniques, are only a few ways to model the test space; the greater the number and variety of models, the better the coverage.

SBTM lends itself well to paired testing. Two sets of eyes together often find more interesting information—and bugs—than two sets of eyes on their own.

Sessions in which 100% of the time is spent on test design and execution are rare. This reconnaissance session is an exception; the focus here is on learning, rather than bug-finding.

Any data files generated or used during the session—in the form of independent reports, program input or output files, screen shots, and so on—get stored in a directory parallel to the library of session sheets.

A test coverage outline is a useful artifact with which to guide and assess a test strategy (the set of ideas that guide your test design), especially one which we're using exploratory approaches. A test coverage outline can be used as one of the inputs into the design of session charters.

```
Prominent Windows:

      Main Table window
      Criteria Weights window
      Option Ratings window
      Documents window
      Start-up window

Managers and Wizards:

      DecideRight Advisor
      Category Label Editor
      Numeric Editor
      Scenario Manager
      Report Generator
      QuickBuild

Decision Elements:

      Language Elements
      Preferences
      Sensitivity Indicators
      Weighting
      Input Options
      Decision Table
      Options Ratings
      Baseline

Interoperability:

      OLE
      Import / Export
      Graphs
      Printing

Risk list for DecideRight:

* It will suggest the wrong decisions.
* People will use the product incorrectly.
* It will incorrectly compare scenarios.
* Scenarios may become corrupted.
* It will not be able to handle complex decisions.

BUGS
-----------------------------------------------
#N/A

ISSUES
-----------------------------------------------
#ISSUE
Manual mentions different platforms (Win 3.1, WFW, and WinNT 3.51) and does
not mention Win2000. We think Win 2000 is important to test on and that the
older OSes are no longer meaningful.

#ISSUE
We did this analysis on Win98.  I have no data to suggest that features may
be different on other operating systems, but I'm not sure about that.
```

A risk list is another useful tool to help guide a test strategy. The risk list can be as long or as short as you like; it can also be broken down by product or coverage areas.

"Issues" are problems that threaten the value of the testing process. Issues may include concerns, requests for missing information, a call for tools or extra resources, pleas for testability. In addition, if a tester is highly uncertain whether something is a bug, that can be reported here.

Example: Session Sheet for a Bug-Finding Session

```
CHARTER
-----------------------------------------------
Explore a decision created with QuickBuild -- the wizard that guides the user
through the options, criteria, and weights needed to calculate the best
decision.

#AREAS
OS | Win98
Build | 1.2
DecideRight | QuickBuild
DecideRight | Report Generator
Strategy | Exploration & Analysis

START
-----------------------------------------------
4/17/01 1:30pm

TESTER
-----------------------------------------------
Jonathan Bach

TASK BREAKDOWN
-----------------------------------------------

#DURATION
short

#TEST DESIGN AND EXECUTION
70

#BUG INVESTIGATION AND REPORTING
20

#SESSION SETUP
10

#CHARTER VS. OPPORTUNITY
90/10

DATA FILES
-----------------------------------------------
food.drd
food.rtf
food2.rtf
food3.rtf

TEST NOTES
-----------------------------------------------

Created a new "decision" that I already knew the answer to: What kind of food
to have for dinner?  I wanted to see if DecideRight could reach the same
conclusion.

Options:
  * American
  * Chinese
  * Mexican
  * Italian
  * Pizza
  * Nothing
```

A single session can cover more than one functional area of the product. Here the testers obtain coverage on both the QuickBuild wizard and the report generator

The goal of any testing session is to obtain coverage—test design and execution, in which we learn good and bad things about the product. Bug investigation (learning things about a particular bug) and setup (preparing to test), while valuable, are interruptions to this primary goal. The session sheet tracks these three categories as inquiry metrics—metrics that are designed to prompt questions, rather than to drive decisions. If we're doing multiple things at once, we report the highest-priority activity first; if it happens that we're testing as we're investigating a bug or setting up, we account for that as testing.

Test notes tend to be more valuable when they include the motivation for a given test, or other clues as to the tester's mindset. The test notes—the core of the session sheet—help us to tell the testing story: what we tested, why we tested it, and why we believe that our testing were good enough.

Information generated from the session sheets can be fed back into the estimation process.

- First, we'll cast a set of charters representing the coverage that we'd like to obtain in a given test cycle. (Let's say, for this example, 80 charters).
- Second, we'll look at the number of testers that we have available. (Let's say 4.)
- Typically we will project that a tester can accomplish three sessions per day, considering that a session is about 90 minutes long, and that time will be spent during the day on email, meetings, breaks, and the like.
- We must also take into account the productivity of the testing effort. Productivity is defined here *the percentage of the tester's time spent*, in a given session, *on coverage*—that is, on test

design and execution. Bug investigation is very important, but it reduces the amount of coverage that we can obtain about the product during the session. It doesn't tell us more about the product, even though it may tell us something useful about a particular bug. Similarly, setup is important, but it's *preparing to test*, rather than *testing*; time spent on setup is time that we can't spend obtaining coverage. (If we're setting up and testing at the same time, we account for this time as testing. At the very beginning of the project, we might estimate 66% productivity, with the other third of the time spent on setup and bug investigation. This gives us our estimate for the cycle:

80 charters x .66 productivity x 4 testers x 3 sessions per day = **10 days**

```
Criteria:
  * price
  * taste
  * convenience
  * last had
  * health

Report notes:

FOOD.RTF

  "Nothing" appears to be the best choice even though my answer was "Pizza."

  ??? How did it reach this calculation?  I would like to devote a session to
this.
  ??? what's the difference between N/A and ??? values
  (see BUG 1 below)

FOOD2.RTF

  * DecideRight showed my 6 choices (options) in order of importance but does
not describe why it ranked them (see BUG below)
  * DecideRight did show my criteria ranked in order, however

FOOD3.RTF

  Created this file because I had a test idea: add some new criteria options
to an existing decision table and re-run the report.

      Result: PASS -- changes get reflected and recalc'ed

      Found a problem in the formatting, though (see BUG 3)

  Test Idea: does eliminating unknown values remove the disclaimer at the top
of the report: ("Warning!  Some elements in the decision table which
generated this report are labeled "To Be Rated" or "Unknown," and it may
therefore be premature to draw conclusions from the data.")

      Result: PASS -- the disclaimer was removed.


OPPORTUNITY: Noticed that pushpin icon on toolbar for decision table does
nothing when no option is highlighted. (see BUG 4 below)

Session interrupted by phone call.  Will pick this up in other session
tomorrow.

Conclusions: I'd like another session or two to learn the algorithm
DecideRight uses to make decisions.  Then I can verify that the report is
accurate.


BUGS
----------
#BUG 1
Not dragging the weight slider for a criteria item leads to an ??? instead of
max "Poor"

Repro:
1 -- launch QuickBuild to create a new decision
2 -- put in some options | Next
```

Exploratory testing, by intention, reduces emphasis on specific predicted results, in order to reduce the risk of inattentional blindness. By giving a more open mandate to the tester, the approach affords better opportunities to spot unanticipated problems.

New test ideas come up all the time in an exploratory testing session. The tester is empowered to act on them right away.

"Opportunity" work is testing done outside the scope of the current charter. Again, testers are both empowered and encouraged to notice and investigate problems as they find them, and to account for the time in the session sheet.

The #BUG tag allows a text-processing tool to transfer this information to a master bug list in a bug tracking system, an ASCII file, or an Excel spreadsheet.

"Some options" might be vague here; more likely, based on our knowledge of the tester, the specific options are be unimportant, and thus it might be wasteful and even misleading to provide them. The tester, the test manager, and product team develop consensus through experience and mentoring on how to note just what's important, no less and no more.

When new information comes in—often in the form of new productivity data—we change one or more factors in the estimate, typically by increasing or decreasing the number of testers, increasing or reducing the scope of the charters, or shortening or lengthening the cycle.


```
3 -- put in some criteria | Next
4 -- when weighing the criteria move on to the Rate Options portion
5 -- don't move the slider for one of the options
6 -- run the report

Result:  Graph shows that value as being ??? instead of "Poor".   Since the
default position of the rating slider is at the end of the Poor scale, I
assumed it would be logged as a maximum "Poor" value, not "unknown".

#BUG 2
Report is missing descriptor for Option section

Repro:
1 -- create a decision using QuickBuild
2 -- File | Generate Report

Result:  The preamble to the list of ranked choices is missed a descriptor
that tells in which order they were ranked.  In the criteria section of the
report, it tells the order: ("The criteria used to evaluate the options were
(in order of importance))."


#BUG 3
Graph labels (y-axis) are cut of if they are longer than 20 characters

Repro:
1 -- create a decision with options that are over 20 characters
2 -- run through QuickBuild with all the defaults
3 -- File | Generate Report

Result: The y-axis labels are truncated.

#BUG 4 OPPORTUNITY
Pushpin toolbar button ("View/edit explanatory text for a decision element")
doesn othing if no option is selected in the decision table

Repro:
1 -- launch a decision table
2 -- click the pushpin icon

Result: No response.

Expected:  Would be helpful if a dialog that tells me I have to select an
option first.

ISSUES
-----------------------------------------------
#ISSUE 1
I'd like another session or two to learn the algorithm DecideRight uses to
make decisions.  Then I can verify that the report is accurate.

#ISSUE 2
What's the difference between N/A and ??? values?
```

Listing all of the possible expectations for a given test is impossible and pointless; listing expectations that have been jarred by a probable bug is more efficient and more to the point.

A step-by-step sequence to perform a test leads to repetition, where variation is more likely to expose problems. A step-by-step sequence to reproduce a discovered problem is more valuable.

Some questions have been raised as to whether exploratory approaches like SBTM are acceptable for high-risk or regulated industries. We have seen SBTM used in a wide range of contexts, including financial institutions, medical imaging systems, telecommunications, and hardware devices.

Some also question whether session sheets meet the standards for the accountability of bank auditors. One auditor's liaison with whom I have spoken indicates that his auditors would not be interested in the entire session sheet; instead, he maintained, "What the auditors really want to

see is the charter, and they want to be sure that there's been a second set of eyes on the process. They don't have the time or the inclination to look at each line in the Test Notes section."

Conclusion

Notebooks have been used by people in the arts, sciences, and skilled professions for centuries. Many exploratory testers may benefit from the practice of taking notes, sketching, diagramming, and the like, and then using the gathered information for retrospection and reflection.

One of the principal concerns of test managers and project managers with respect to exploratory testing is that it is fundamentally unaccountable or unmanageable. Yet police, doctors, pilots, lawyers and all kinds of skilled professions have learned to deal with problem of reporting unpredictable information in various forms by developing note-taking skills. Seven years of positive experience with session-based test management suggests that it is a useful approach, in many contexts, to the process of recording and reporting exploratory testing.

Thanks to Launi Mead and Doug Whitney for their review of this paper.

Install Risk Catalog

Functional suitability

- Installer lacks modern, expected features
  - no uninstall
  - no custom install
  - no partial install (“add”)
  - no upgrade install

Reliability

- Intermittent failure

Fault tolerance/recoverability

- Can’t back up
- Can’t abort
- No clean up after abort
- Mishandled read error
- Mishandled disk full error

Correctness

- Wrong files installed
  - temporary files not cleaned up
  - old files not cleaned up after upgrade
  - unneeded file installed
  - needed file not installed
  - correct file installed in the wrong place
- Wrong INI settings/registry settings
- Wrong autoexec/config settings
- Files clobbered
  - older file replaces newer file
  - user data file clobbered during upgrade

Compatibility

- Installer does not function on certain platforms
- Other apps clobbered
- HW not properly configured
  - HW clobbered for other apps
  - HW not set for installed app
- Screen saver disrupts install
- No detection of incompatible apps
  - apps currently executing
  - apps currently installed

Efficiency

- Excessive temporary storage required by install process

Usability

- Installer silently replaces or modifies critical files or parameters
- Install process is too slow
- Install process requires constant user monitoring
- Install process is confusing
  - UI is unorthodox
  - UI is easily misused
    - Messages and instructions are confusing
    - Mistakes during install process readily cause loss of effort

TNT QA Task Analysis

BC4.0 & BP7.0

7/12/92

QA Requirements Summary:

Tool	Popularity	Rate of Change	Complexity	Existing automation	Required Testing*
TD32	**High**	**High**	**High**	None	**Extensive**
TDX	**High**	**High**	**High**	Minimal	**Extensive**
TDW	**High**	**High**	**High**	Moderate	**Extensive**
TDV	**High**	**High**	**High**	Moderate	**Extensive**
TD286	**High**	Low	**High**	Moderate	Moderate
TD386	**High**	Low	**High**	Moderate	Moderate
TD	**High**	Low	**High**	Moderate	Moderate
GUIDO	**High**	**High**	**High**	Minimal	**Extensive**
TPROF	Moderate	Moderate	**High**	None	**Moderate**
TPROFW	Moderate	Moderate	**High**	None	**Moderate**
TF386 (TFV)	Low	Low	**High**	Minimal	**Moderate**
TFREMOTE	Low	Low	Moderate	None	**Minimal**
TDREMOTE	Moderate	Low	Moderate	None	**Minimal**
WREMOTE	Low	Low	Moderate	None	**Minimal**
WRSETUP	Low	Low	Low	None	None
TDRF	Moderate	Low	Low	None	None
TDUMP32	Moderate	Moderate	Low	None	**Minimal**
TDUMP	Moderate	Low	Low	None	None
TDINST	Moderate	Moderate	Low	Minimal	Minimal
TDINST32	Moderate	**High**	Low	None	**Minimal**
TDSTRIP	???	Moderate	Low	None	**Minimal**
TDMEM	???	Low	Low	Minimal	None
TDDEV	???	Low	Low	Minimal	None
TDH386.SYS	**High**	Low	Low	Moderate	Moderate
TDDEBUG.386	**High**	Low	Low	None	**Minimal**
Examples	???	Moderate	Low	None	**Minimal**
TASM & tools	Moderate	Low	**High**	Moderate	Moderate

*
Items are boldfaced where the existing automation and beta testing will have to be augmented by new automation and hand testing.

Task Sets (? denotes unstaffed):

Guido Testing
(General Testing Tasks)
Produce feature outline
Produce sign-off checklist
Complete smart-script system version 1.0
Analyze hard mode vs. soft mode Integrate 100 applications into smart-script system

TDX Testing
(General Testing Tasks)
Produce sign-off checklist
Maintain communication between Purart and Gabor
Learn about DPMI
Test real mode stub
Test remote debugging

? TD32 Testing (Windows)
(General Testing Tasks)
TDINST32
Produce sign-off checklist
Learn WIN32s platform
Determine Windows NT dependencies
Track changes in NT and WIN32s Track differences between Microsoft Win32s & Rational

? TD32 Testing (DPMI32)
(General Testing Tasks)
TDUMP32
Produce debugger example for doc.
Produce sign-off checklist
Learn DPMI32 platform
Track development of DPMI32
Coordinate testing w/DPMI32 testers

? TD/TDV Testing
(General Testing Tasks)
TD286
TD386
TDH386.SYS
TDSTRIP
TDUMP
TDMEM
TDDEV
TDRF
TDREMOTE
TDINST
Produce sign-off checklist

? TDW Testing
(General Testing Tasks)
TDDEBUG.386
WREMOTE
WRSETUP
Produce sign-off checklist
Track SVGA DLL development

? Profiler Testing
(General Testing Tasks)
TPROF
TPROFW
TF386
Produce sign-off checklist
Produce feature outline
Review automation coverage
Verify timing statistics
Collect very large applications
Identify & support in-house users
Identify & support key beta sites
Develop TFSMERGE program

? Automation1 (lead)
Produce ~600 new tests to satisfy test matrix
Produce 16-bit debugger feature outline
Assist in producing overall test matrix
Produce next generation C++-based test control system
Produce feature coverage viewer program
Produce Monkey-based acceptance suite for Purart
Convert smart-script system to Alverex tools
Maintain DCHECK & TCHECK

Automation2 (support)
Execute all automation and generate reports
Fix tests that break in old test system (500 total)
Generate BTS reports weekly
Adapt test system to OS/2
Adapt test system to NT
Produce ~600 new tests to satisfy matrix
Recompile test attachments with new compiler
Perform compatibility testing

Diablo1 (process control)

Diablo2 (data inspect)

Diablo3 (general functions)

TASM Testing

PROCHAIN ENTERPRISE

Scenario Test Plan

Overview

Scenario testing is about how the product behaves when subjected to complex sequences of input that mirror how it was designed to be used, as well as how it might realistically be misused. A scenario, in this context, is a story about how the product might be used. Through scenario testing we hope to find problems that lie in the interactions among different features, and problems that are more important because they occur during particularly common or critical flows of user behavior.

This document describes an exploratory form of scenario testing. Our documentation philosophy is based on that of the *General Functionality and Stability Test Procedure* (see http://www.satisfice.com/tools/procedure.pdf) used by Microsoft's compatibility test group and in Microsoft's *Certified for Windows* logo program. In this process, scenario test charters are produced, and those charters (which could also be described as very high level test procedures) are used to guide scenario tests performed by experienced users.

Status: We have collected a lot of scenario ideas and data. We are about a third of the way through the process of documenting it, but we have already begun the test process.

Scenario Charter Design Process

Good scenario test design requires knowledge of the purposes that the product serves and the context in which it is used. So, we used two Prochain staff consultants and the author of the user documentation as domain experts to help produce the scenarios. Scenario design included these activities:

- **User documentation exhibits.** Review documentation provided by friendly customers and the development team. Such documentation describes how Prochain Enterprise is used by various kinds of users, including step-by-step instructions for updating data in the system.
- **Facilitated brainstorm with domain experts.** Review goals and patterns of scenario testing, then brainstorm test ideas. These ideas may include standalone elements to be incorporated into scenarios, as well as fully worked scenario scripts, with variations.
- **Chartered exploratory test sessions.** Pick a couple of mainstream scenario ideas and conduct exploratory test sessions, using domain experts as testers. In these sessions, follow a scenario theme, developing it further while recording what each tester did using both automatic recorders and personal observation. All the testers should use the same database to gain the benefit of implicit multi-user testing. While some testers coordinate with each other to flesh out the scenarios, others assist in taking notes or investigating problems.
- **Scenario refinement.** Once scenarios are roughed out, discuss, prune, and extend them. Look for missing elements, and compare them with user documentation exhibits.
- **Function tracing.** Compare the scenarios to the features of the product to assure that we have scenarios that, in principle, cover all the functions of the product.

Scenario Design Elements

During our design process, various elements of scenarios were identified, and we used these ideas to design the present scenario set. Further development of the scenarios might benefit by taking these ideas into account and extending them.

Activity Patterns

These are used as guideword heuristics to elicit ideas for deepening and varying the activities that constitute the scenario charters.

- *Tug of war; contention.* Multiple users resetting the same values on the same objects.
- *Interruptions; aborts; backtracking.* Unfinished activities are a normal occurrence in work environments that are full of distractions.
- *Object lifecycle.* Create some entity, such as a task or project or view, change it, evolve it, then delete it.
- *Long period activities.* Transactions that take a long time to play out, or involve events that occur predictably, but infrequently, such as system maintenance.
- *Function interactions.* Make the features of the product work together.
- *Personnas.* Imagine stereotypical users and design scenarios from their viewpoint.
- *Mirror the competition.* Do things that duplicate the behaviors or effects of competing products.
- *Learning curve.* Do things more likely to be done by people just learning the product.
- *Oops.* Make realistic mistakes. Screw up in ways that distracted, busy people do.
- *Industrial Data.* Use high complexity project data.

Scenario Personnas

- *Individual Contributors.* Individual contributor scenarios involve updating tasks and viewing task status.
- *Analysts (e.g. critical chain experts, resource managers, consultants).* Analyst scenarios focus on viewing and comparing tasks and projects, using the reporting features, and repeatedly popping up and drilling down.
- *Managers (e.g. task managers, project managers, senior management).* Management scenarios involve analysis, but managers also coordinate with individual contributors, which leads to more multi-user tests. Managers update buffers and may download schedules and rewire them.
- *System Administrators.* System administration scenarios involve the creation and removal of users, rights setting, system troubleshooting and recovery.

Test Dimensions

To test Prochain Enterprise effectively, all of the following variables must be considered, controlled and systematically varied in the course of the testing. Not all scenarios will specify all of these parts, but the testers must remain aware of them as we evaluate the completeness and effectiveness of our work.. Some of these are represented in the structure of the scenario charters, others are represented in the activities.

- *Date.* Manipulation of the date is important for the longer period scenario tests. It may be enough to modify the simulation date. We might also need to modify the system clock itself. *Are we varying dates as we test, exploring the effects of dates, and juxtaposing items with different dates?*

- *Project Data.* In any scenario other than project creation scenarios, we need rich project data to work with. Collect actual industrial data and use that wherever possible. *Are we using a sufficient variety, quantity and complexity of data to approximate the upper range of realistic usage?*

- *User Data.* In any scenario other than system setup, we need users and user rights configured in diverse and realistic ways, prior to the scenario test execution. *Are enough users represented in the database to approximate the upper range of realistic usage? Is a wide variety of rights and rights combinations represented? Is every user type represented?*

- *Functions.* Capability testing focuses on covering each of the functions, but we also want to incorporate every significant function of the product into our set of scenario tests. This provides one of the coverage standards we use to assess scenario test completeness: *Is every function likely to be visited in the course of performing all the scenario tests?*

- *Sequence.* The specific sequence of actions to be done by the scenario tester is rarely scripted in advance. This is because the sheer number of possible sequences, both valid and invalid, is so large that to specify particular sequences will unduly reduce the variety of tests that will be attempted. We want interesting sequences, and we want a lot of different sequences: *Are testers varying the order in which they perform the tasks within the scenario charters?*

- *Simultaneous Activity and States.* Tests may turn out differently depending on what else is going on in the system at any given moment, so the scenario tests must consider a variety of simultaneous event tests, especially ones involving multi-user contention. *Are the testers exploring the juxtaposition of potentially conflicting states and interactions among concurrent users?*

- *System Configuration.* Testing should occur on a variety of system configurations, especially multi-server configurations, because the profile of findable bugs may vary widely from one setup to another. *Are scenario tests being performed on the important configurations of Enterprise?*

- *Oracles.* An oracle is a principle or mechanism by which we recognize that a problem has occurred. With a bad oracle, bugs happen, but testers don't notice them. Domain experts, by definition, are people who can tell if a product is behaving reasonably. But sometimes it takes a lot of focus, retesting, and special tooling to reliably detect the failures that occur. *For each scenario, what measures are testers taking to spot the problems that matter?*

- *Tester.* Anyone can perform scenario testing, but it usually takes some domain expertise to conceive of activities and sequences of activities that are more compelling (unless it's a Learning Curve scenario). Different testers have different propensities and sensitivities. *Has each scenario test been performed by different testers?*

This is our first cut at a fundamental set of scenario themes. Each sub-theme listed below stands alone as a separate scenario test activity. They can be performed singly, or in combination by a test team working together.

- *Project Update*
  - **UP1:** Check tasks and update.
  - **UP2:** Check status and perform buffer update.
  - **UP3:** Check out a project and rewire dependencies.
  - **UP4:** Troubleshoot a project.
- *Project Creation*
  - **CR1:** Add projects, finish projects, observe impact.
  - **CR2:** Set project views, attachments, and checklists.
- *System Administration*
  - **SA1:** Administration setup and customization.
  - **SA2:** Rescale the configuration.

Scenario Testing Protocol and Setup

Mission	Find important bugs quickly by exploring the product in ways that reflect complex, realistic, compelling usage.
Testers	- As a rule, the testers should understand the product fairly well, though an interesting variation of a scenario can be to direct a novice user to learn the product by attempting to perform the scenario test. - The testers should understand likely users, and likely contexts of use, including the problems users are trying to solve by using the product. When testers understand this, scenario testing will be a better counterpoint to ordinary function testing. - The testers should have the training, tools, and/or supervision sufficient to assure that they can recognize and report bugs that occur.
Setup	- Select a user database & project database *that you can afford to mess up* with your tests. - Assure that the project database has at least two substantial projects and program in it, preferably more. The projects should include *many* tasks, statuses of green/yellow/red, and multiple buffers per project. - Tasks should have *variety*, e.g. short ones, long ones, key tasks, non-key tasks, started, not-started, with and without attachments and checklists. - Set the simulation date to intersect with the project data that you are using. - Fulfill the setup requirements for the particular scenario test you are performing.
Activities	In exploratory scenario testing, you design the tests as you run them, in accordance with a *scenario test charter*: ❑ Select a scenario test charter and spend about 90 minutes testing in accordance with it. ❑ Perform the activities described in the test charter, but also perform variations of them, and vary the sequence of your operations. ❑ If you see something in the product that seems strange and may be a problem, investigate it, even if it is not in the scope of the scenario test. You can return to the scenario test later. ❑ Incorporate micro-behaviors freely into your tests. Micro-behaviors include making mistakes and backing up, getting online help in the middle of an operation, pressing the wrong keys, editing and re-editing fields, and generally doing things imprecisely— the way real people do. ❑ Do things that should cause error messages, as well as things that should not. ❑ Ask questions about the product and let them flavor your testing: What will happen if I do *this?* Can the product handle *that?* ❑ Consider working with more than one tester on more than one scenario. Perform multiple scenarios together. ❑ Remember to advance the timeline periodically, either using the simulation date or using the system clock.
Oracle Notes	- Review the oracle notes for the scenario charter that you are working with. - Review and apply the HICCUPP heuristics. - For each operation that you witness the product perform, ask yourself how you know that it worked correctly. - Perform some operations with data chosen to make it easy to tell if the product gave correct output. - Look out for progressive data corruption or performance degradation. It may be subtle.
Reporting	- Make a note of anything strange that happens. If you see a problem, briefly try to reproduce it. - Make a note of obstacles you encountered in the test process itself. - Record test ideas that come to you while you are doing this, and pass them along to the test lead.

UP1: "Check tasks and update"

Theme	You are an individual contributor on a project. You have tasks assigned to you. Check your tasks and update them. Check the status of tasks that gate the ones you are responsible for.
Setup	- Assure that your user account(s) are set up with rights to access a project that has *many* tasks assigned to it.
Activities	❑ Go to Tasks panel and filter tasks for ones assigned to you. (Alternatively, filter in other ways such as by project or by incomplete tasks; and choose a way to sort) ❑ Select one of the task list views and visit each task. Set the task filter to show, at least: actual start, total duration, and remaining duration. ❑ For some tasks, view details, checklists, and attachments. ❑ Update each task in some way, including: - No update - "Mark as Updated" - Shorten duration remaining - Set remaining duration to zero; or "Mark as Completed" - Increase duration remaining - Provide comments; update checklist - Undo some updates ❑ Refilter to see more tasks. Find tasks that feed into or lead from your tasks. Update some of those tasks.
Oracle Notes	- View updated tasks *prior to buffer* update to verify they have been updated properly. - View updated tasks *after buffer update* to verify they are correct. - Verify that an updated task says "started" or where applicable verify that it has become a key task or that it has ceased to be a key task. - Determine the total number of tasks visible within MS project file, and verify all are visible in Enterprise.
Variations	- *USER DATA:* Restrict the rights of the user account to the maximum degree while still being able to perform the activity. - *TUG OF WAR:* log in as a second user and re-update the same tasks, or cancel updates; log in as the same user as if you forgot you already had another window open, then make changes in both windows. - *OOPS:* update the wrong task and then undo the update; update a task, wait for buffer update, then realize you screwed up and try to fix it. - *INTERRUPTION:* Try to make updates while a buffer update is going on. - *LIFECYCLE:* Update a fresh task, update it several more times, advancing the simulation date, then mark it as completed. Do that for an entire project. Mark all tasks as completed.

UP2: "Check status and perform buffer update"

Theme	You are a project manager. You need to update your project to prepare your weekly report on project status.
Setup	- Log in with a user account set up with project manager rights. - Buffer consumption for one of the projects should ideally be in the yellow or red. - At least some of the projects should have multiple project buffers.
Activities	❑ View the Standard Projects Status Chart (or custom chart), filter on a set of projects (and turn on name labels). Start a second session in a window next to the first one (log in as the same user), and filter for the same project set. Now you have two project status charts that you can compare. ❑ Pick one project as "yours". Now, compare status history of your project to others. Explore the other project details in any way necessary to account for the *differences* in status. ❑ View all impact chains for your project, and for some of those tasks: - view task details - view task links - view task load chart ❑ If other testers are making task updates during your test session, review those changes and modify some of them, yourself. Otherwise, make at least a few updates of your own. ❑ Advance the clock by a few days, update buffers on your project and view again the status chart and impact chains, then advance the clock again by another few days. ❑ Search for all project tasks that have not been updated in more than a "week" (i.e. since the test began). Update some of them, then perform another buffer update and view status history for that project.
Oracle Notes	- View updated tasks *prior to buffer* update to verify they have been updated properly. - View updated tasks *after buffer update* to verify they are correct. - Verify that an updated task says "started" or where applicable verify that it has become a key task or that it has ceased to be a key task. - Determine the total number of tasks visible within MS project file, and verify all are visible in Enterprise. - Verify the reasonableness of the impact chains, updates to the impact chains, and status history.
Variations	- *USER DATA:* superuser "accidentally" changes your user permissions during the test so that you can no longer change your own project. - *TUG OF WAR:* a second user logs in and checks out the project that you are analyzing, locking it. - *OOPS:* update project notes and comments in the wrong project, and try to remove them and apply them to the right project. - *INTERRUPTION:* Periodically click on the printer icon.

UP3: "Check out a project and rewire dependencies"

Theme	You are a project manager. Your project has changed as a result of new technology or new resources, and the current network needs to be updated.
Setup	- Log in with a user account set up with project manager rights.
Activities	❑ Pick a project as "yours". Check out the project file to your local hard drive. ❑ Update the project network in MSP, do a selection of the following: - Add new tasks that have starting dates before the present date, some that span the present date, and some that end in the future. - Add new tasks that are not on the critical chain, and some that are. - Delete some tasks. - Modify data in custom fields. - Change some of the task linkages. - Reassign resources; Overload some resources. - If the project has one endpoint, add a second endpoint; if it has two multiple endpoints, remove all but one. *(remember to keep track of the changes you make!)* ❑ Check the project back into PCE, and update buffers. ❑ View all impact chains for your project, and for the tasks and chains that you modified: - view task details - view task links - view task load chart
Oracle Notes	- The new network's info are correctly represented in PCE: - buffer consumption - impact chain - key tasks - resources and managers - On check-in PCE should force a buffer update.
Variations	- *TUG OF WAR:* A second user logs in and checks in the project while you are changing it. - *OOPS:* Check in the wrong project file, and then try to recover. - *OBJECT LIFECYCLE:* Rewire the project several times, interspersing that with UP1 an UP2 scenarios. Then complete all tasks.

OWL Quality Plan

Final

This document incorporates all previous Elvis quality assurance documents. It is an analysis of the tasks necessary to assure quality for Elvis. It has been reviewed by Tech. Support, and reflects the concerns of our customers.

This document includes the following sections:

- Risk and Task Correlation
- Component Breakdown
- Ongoing Tasks

Resource loading and open issues are not included, due to time constraints, and the need for broader review by management.

Risk and Task Correlation

This table relates risk areas to specific quality assurance tasks. Any tasks listed on the right which are not completed will increase the likelihood of customer dissatisfaction in the associated risk area on the left.

Source Code Usability	• Review code for comments, style, formatting, and comprehensibility. • Review makefiles for simplicity, documentation, and consistency.

Performance	• Benchmark performance of low level encapsulation and high-order functionality versus
	  • OWL 1.0x
	  • MFC
	  • Native Windows apps
	• Actively solicit Beta tester feedback, design questionnaire, tabulate/analyze results.

Internationalization	• Verify international enabling of the following:
	  • Stored strings (window titles, diagnostics, etc.)
	  • Menus items and accelerators
	  • Cutting and pasting text (clipboard support)
	  • Printing
	  • Localized versions of common dialogs
	  • Status line code
	  • Input validation (proper uppercasing, etc.)
	  • filenames/streaming

Design Quality	• Inspect code for appropriate use of C++ idioms.
	• Participate in discussions to promote:
	  • Design simplicity
	  • Backward compatibility
	  • Appropriate feature set
	  • Flexibility for future technologies

Documentation Quality *Reference Guide*	• Confirm API coverage with latest available header files. • Check completeness of information for each API, member function, and data item. • Review material for overall usability/organization.

Programmer's Guide	• Check for missing pieces:
	  • Versus MFC –
	  • Versus Petzold (native Windows)
	  • Versus our provided examples
	  • Revealed by beta survey feedback
	  • RTL/Classlib functionality used by Elvis
	  • C SDK methods compared with Elvis methods
	• Review example code versus:
	  • Code style/readability/comprehensibility
	  • Compile-time errors/warnings
	  • Run-time bugs
	• Review material for overall usability/organization.

Tutorial	• Actively solicit feedback from neophyte Elvis users.
	• Review example code versus:
	  • Code style/readability/comprehensibility.
	  • Compile-time errors/warnings.
	  • Run-time bugs.

Application size and efficiency	• Benchmark Elvis size (DGROUP, .EXE) and performance vs.:
	  • Elvis 1.0x
	  • MFC
	  • Native Windows apps
	• Check diagnostics
	• Measure effect of varying levels of diagnostics
	• Determine optimum/shipping versions of final vs. 'debug' libraries, re: size/efficiency
	• Actively solicit Beta feedback from
	  • Power Users (substantial/industrial strength apps.)
	  • Users of C++ that don't tend to write "optimal" code (e.g., reviewers)

Debugger support	• Review comprehensiveness and appropriateness of diagnostics on a class by class basis
	• Verify debugger support for
	  • Special Elvis needs: entry point/Winmain issues, Elvis diagnostics, etc.
	  • Any debugging problems highlighted by Elvis: heavily templatized code, exceptions, RTTI, linker capacity, etc.
	• Lobby for debugger features needed to enhance Elvis debugging, e.g., memory mgmt. diagnostics, heap walking capability, etc.

Portability across platforms, APIs, and compilers	• Review Elvis source to assure appropriate use of APIs::
	• `#ifdef` or remove Win16-specific calls
	• `#ifdef` full Win32-specific calls
	• `#ifdef` Win16 calls which have better Win32/s equivalents
	• Execute test suites to verify that examples and other suites produce the same output for both static and dynamic libs.
	• Investigate the following C++ Compilers for Elvis compatibility:
	  • Symantec
	  • MetaWare
	  • Microsoft
	  • CFront
	• Execute test suites to verify that examples and other suites produce appropriate output for the following (using debug kernel):
	• Win 3.1
	• Win32s on Win 3.1
	• Win32/s on Windows NT
	• Win 3.1 on Windows NT
	• Win 3.1 on OS/2
	• Investigate Elvis compatibility using Mirrors on OS/2.

High-order functionality
System level
	• Review specifications to assure that the following functionality is supported
	  • OLE
	  • VBX
	  • GDI
	  • BWCC
	  • CTRL3D
	• Track support issues for 3rd party:
	  • Frameworks
	  • Class libraries (Rogue Wave, etc.)
	  • Custom control (widget) collections
	• Track interoperability issues for Borland products:
	  • Class libraries (Classlib, RTL iostreams, etc.)
	  • Engines (Pdox, BOLE2, etc.)
	  • Internal and external tools (WMonkey, WinSight, Tarzan, Lucy, CBT, etc.)

Feature level	• Verify that examples exist that use features of the 32bit platforms and that include the following functionality:
	  • Event response tables to replace DDVTs
	  • Windows' resources from multiple DLLs; TLibManager
	  • Document View model
	  • OLE DocFile support
	  • Common dialogs
	  • Clipboard support
	  • Floating palette
	  • Window decorations/gadgets (tool bars/status bars)
	  • Input validation support
	  • Printer support
	  • Use of C++ exceptions
	  • Menus (including OLE 2.0 support)
	  • GDI (fonts, brushes, pens, palettes, bitmaps, regions, icons, cursors, DIBs, complete device context encaps.)
	  • Virtual listboxes (1,000,000,000 items)
	  • Edit control without limits
	  • Outliner/Tree structure listbox
	  • Edit control that will take multiple fonts
	  • Print Preview
	  • Edit control like QPW's
	  • Gauges, sliders, spin buttons, split panes
	  • Example(s) showing use of ODAxxxxx (OwnerDrawAccess APIs)
	  • Workshop aware custom controls (there's already a hack on CIS)
	  • OWL custom control(s) that are usable by 'C SDK' style applications

Low-level API encapsulation	• Review message response macros for coverage. • Verify that all appropriate APIs (i.e., OS features) are encapsulated. • Compare item-by-item to MFC and other competitors • Verify that API functionality is fully accessible and fully usable. • Check internal data structures for completeness. • Verify consistency of Elvis abstractions (i.e., compared to the native API parameter order, data types, etc.). • Actively solicit feedback on ease-of-use/friendliness of enabling layer Elvis API.

Backward compatibility and upgradeability	• Assure that the BC4 toolset will work with OWL 1.0x
	• Assure that OWL 1 apps are upgradeable to Elvis vis-a-vis:
	  • Documentation (usability testing, beta banging, careful inhouse review)
	  • Automated conversion tool works intuitively
	  • Usability and documentation of design changes
	  • A comparison of 'major' techniques used in OWL 1.0x with their current method in Elvis (Are they unnecessarily different? Are they so much better that they're worth the pain to switch? Are the above questions/answers/design decisions fully doc'ed?)

Reliability	• Measure code coverage of examples to determine what should be stressed by new tests.
	• Create or collect special test code, including at least one large-scale omnibus application.
	• Create and maintain smoke tests runnable by Integration.
	• Build OWL library, after each delivery that has changes in source or include files, for:*
	  • 16bit small static
	  • 16bit medium static
	  • 16bit large static
	  • 16bit large DLL
	  • 32bit flat static
	  • 32bit flat DLL
	  • All of the above in diagnostic/debugging mode.
	• Build selected models with -Vf, -O2, -xd, -3, -dc and -po:‡
	  • 16bit large/medium static (switch every other time between medium and large)
	  • 16bit large DLL
	  • 32bit flat fully optimized for speed and/or size (if not already delivered that way)
	• Verify that user built libs are identical to 'delivered' libs (except paths and time stamps).
	• Build all examples in all models listed above and run automated regressions
	• Verify that OWLCVT converts its test suite correctly.

† These first 12 will all be delivered to customers, on CD-ROM, the first 6, at least, on diskette.
* The following configurations may also be delivered on CD-ROM, if sufficient testing can be done.

Component Breakdown

This is a breakdown of OWL components to a reasonable granularity:

1. **TEventHandler**
2. **TStreamable**
3. **TModule**
    - 3.1. TApplication
    - 3.2. TLibManager
    - 3.3. TResId
    - 3.4. TLibId
4. **TDocManager**
5. **TDocTemplate**
6. **TDocument**
    - 6.1. TFileDocument
    - 6.2. TDocFileDocument
7. **TView (TEditSearch and TListBox parentage)**
8. **TWindow**
    - 8.1. TDialog
        - 8.1.1. TInputDialog
        - 8.1.2. TPrinterDialog
        - 8.1.3. TCommonDialog
    - 8.2. TControl
        - 8.2.1. TSScrollBarData
        - 8.2.2. TScrollBar
        - 8.2.3. TGauge
        - 8.2.4. TGroupBox
        - 8.2.5. TStatic
        - 8.2.6. TButton
        - 8.2.7. TListBox
    - 8.3. TMDIClient
    - 8.4. TFrameWindow
        - 8.4.1. TMDIChild
        - 8.4.2. TMDIFrame
        - 8.4.3. TDecoratedFrame
        - 8.4.4. TDecoratedMDIFrame
    - 8.5. TLayoutWindow
    - 8.6. TClipboardViewer
    - 8.7. TKeyboardModeTracker
    - 8.8. TFloatingPalette
    - 8.9. TGadgetWindow
9. **TScrollerBase**
    - 9.1. TScroller
10. **TValidator**
11. **TPrinter**
12. **TPrintout**
13. **TGadget**
14. **TException**
15. **TMenu**
16. **TClipboard**

17. **TGdiBase**
    - 17.1. TGDIObject
        - 17.1.1. TRegion
        - 17.1.2. TBitmap
        - 17.1.3. TFont
        - 17.1.4. TPalette
        - 17.1.5. TBrush
        - 17.1.6. TPen
    - 17.2. TIcon
    - 17.3. TCursor
    - 17.4. TDib
    - 17.5. TDC
        - 17.5.1. TWindowDC
        - 17.5.2. TPaintDC
        - 17.5.3. TCreatedDC
        - 17.5.4. TMetafileDC
18. **TPoint**
19. **TRect**
20. **TMetaFilePict**
21. **TDropInfo**
22. **TResponseTableEntry**
23. **TClipboardFormatIterator**
24. **TLayoutMetrics**
25. **Diagnostics support**
26. **Streaming/object persistence support**
27. **Error handling & exceptions**
28. **BOLE2 client/container support**
    - 28.1. Elvis support classes
    - 28.2. BOLE2.DLL component
    - 28.3. ObjectPort interface class
29. **VBX support classes**
30. **OWLCVT porting tool**
    - 30.1. DDVTs to response table entries conversion
    - 30.2. Class name and other text substitutions
31. **Makefiles**
    - 31.1. Library source
    - 31.2. Examples
32. **Examples**
    - 32.1. Large scale (large/complex/high-order feature set)
    - 32.2. Miscellaneous (small size/low-level feature set)
    - 32.3. Non-shipping (but may move into above categories)
33. **Documentation**
    - 33.1. Programmer's Guide
    - 33.2. Reference Guide
    - 33.3. Tutorial
    - 33.4. Online Doc Files
    - 33.5. Online Help

PRODUCT

Issues

COMPANY	Send ST Labs information on how dictation is supposed to be done. HOW2USE.DOC contains no information on dictation protocol.
COMPANY	When will user documentation be available—even in half-baked, development form? It will have to exist in some form months before ship.
COMPANY	Inform ST Labs as to what paper items are included as part of the product.
COMPANY	Send ST Labs the second context for installation testing (for step 2).
COMPANY	What files represent the selection? We need to know the actual filenames, in order to transfer testing from one computer to another without retraining. We understand that we are not to test selections per se.
COMPANY	Is the 100mb of disk space needed for swap files during training over and above the 150mb needed for the sound files and the 50mb needed for the software? Is the minimum total space needed in order to install and train 300mb?
COMPANY	When self-diagnostics are implemented, alert ST Labs and send them information on how they work.
COMPANY	Need CD of standard sound files (male, female)
COMPANY	Determine how existing Word macros are to be integrated with speech aware macros in the same template.
COMPANY	Define subset of functionality testing for use in interoperability tests.
COMPANY	Define subset of functionality testing for use in hardware compatibility tests.
COMPANY	It would help ST Labs (and COMPANY) do better testing for less money if they knew more specifically who are the intended users/groups.
COMPANY	COMPANY to specify their expectations regarding test documentation deliverables with respect to each test task.
COMPANY	Implement backup procedure for speech-critical data files?
ST Labs	Examine the COMPANY bug database for testing insights.
ST Labs & COMPANY	Confirm with COMPANY that UGC will summarize or manage the beta testing feedback, beyond reporting specific bugs. (e.g. collecting information about requested features regarding things like a spelling mode or vocabulary addition mode)

Administration

Leads

ST Labs

1st level: Ken

2nd level: Jim

COMPANY

1st level: Andreas

2nd level: Werner

Facilities

We are acquiring 3 new high-end computer systems on which to test.
We have acquired directional headsets for use in training.

Staff

One test lead and two testers (male and female) during the first step.

Schedule

See the bid for details.

Communication & Deliverables

Build Transfer

Builds will be transferred via ftp.stlabs.com
COMPANY will send ST Labs one new build per week.

Status Reporting

Ken will make daily status reports, weekly summary reports, and a project summary report at the conclusion of the project.
Status reporting will be done via email.

Bug Reporting

Bug reports will be submitted daily via Reachout, directly to the COMPANY DCS bug database.

Test Techniques

Stakeholders

Users (represented by the beta testers)
Andreas (mediates with other sources at COMPANY)
ST Labs (our opinions about the functionality are invited)

Specifications

Functional specification
Windows Interface Guidelines (for Win95 compliance issues)
User Documentation (not available as of 10/4/96)

Risk	Strategy
Windows Compliance *Description:* As a Windows 95 product, it should conform to the Win95 logo requirements and interface guidelines.	*ST Labs* • We can do *basic* Win95 UI conformance testing and we can review the Win95 logo requirements and advise you of possible issues, but there is not enough time in the plan for comprehensive testing in either area. •
General Functionality • Description: The product should function in substantial conformance to the Product Requirements Specification and user documentation	*ST Labs & COMPANY* • exploratory testing • documentation-based testing • specification-based testing • scenario testing • input domain testing •
HW Compatibility *Description:* As a product to be deployed in an open environment, it must be operable with a variety of popular hardware platforms and peripherals.	*Beta tester* • Verify that we have the config. info on each beta tester. *ST Labs* • (include configurations) • Microphones • Sound cards • Systems *COMPANY* • (include configurations) •
Installability *Description:* Since the product will be installed by untrained users, it must be a safe and simple process. (Installability testing does not include the training process.)	(Installability testing does not include the training process.) *ST Labs & COMPANY* • monitor beta testers • clean install testing • upgrade install testing • uninstall testing • installing a new context

SW Compatibility
Description:
As a product to be deployed in an open environment, it must be operable with a variety of popular software products.
	ST Labs
	• Quick dictation interference test
	• Not enough time to test with NT network.
	• Applications that may also use SoundBlaster
	• Netscape
	• Exchange clients (MSMail, Schedule Plus)
	• SAM virus clinic
	• Basic interoperability testing (there is not enough time to do comprehensive testing, here)
	  • non-speech-aware dictation clients
	  • notepad
	  • Ami Pro
	  • Word [1]

[1] Changes in strategy from the 9/19 version of the test plan are highlighted in dark gray.

	COMPANY • Quick dictation interference test • Exchange clients (MSMail, Schedule Plus) • Defrag. utility • Macafee virus scanner • Novell network • NT network • Interoperability testing • non-speech-aware dictation clients • Wordperfect • Word •
System-Level Error Handling *Description:* The product should handle incorrect input or other fault conditions, especially ones the user is most likely to encounter, consistently and gracefully.	*ST Labs* • Exploratory testing • Monitor beta testers • Not enough time for special stress testing and invalid data testing. *COMPANY* • Error testing • Stress testing • Invalid data testing
Data Integrity & Recoverability *Description:* Because the data generated and managed in the course of training and using the system is so vital to its operation, the system should recognize and/or allow recovery from data corruption.	*ST Labs* • Report obvious data corruption during other testing • There is not enough time to do recoverability testing. *COMPANY* • Recoverability testing • bad selection • delete the files • replace the files with dummies • 1. start dictation session using selection; 2. delete the selection from control module; 3. reopen or return to dictation. • bad ARF • power failure during dictation session • power failure during training (ARF) • Test dictionary reorganization at the 64K word limit.

Memory/Mass Storage *Description:* Users may experience failures associated with the large amount of internal memory and mass storage required for this product. • *efficiency:* how files are stored and cleaned up. • *reliability:* what happens under low memory or disk space conditions. Memory leaks. • *usability:* how do users know when and how to delete files or optimize their systems.	*ST Labs* • exploratory testing • documentation-based testing • documentation testing • monitor beta testers • all stress testing is the responsibility of COMPANY *COMPANY* • stress testing • simultaneous applications low disk space and memory configs. • long dictation sessions • large number of corrections • add lots of words in a session
Performance *Description:* Because the usability of the system is strongly dependent on system performance, this performance should be measured and monitored. *Performance dimensions:* • during initial acoustic adaptation • during dictation • language model adaptation (shortterm & long-term) • during further acoustic adaptation *Performance degradation (due to):* • lack of disk space or internal memory • dictation session duration • dictation file size • number of corrections • size of dictionary	*ST Labs* • Qualitative performance testing of framework applications (MIP, unless critical) *COMPANY* • Performance testing of speech recognition technology
Usability *Description:* Users may find this product hard to learn and frustrating to use. *Factors:* • We know very little about the target market and typical user. • The product requires that the user adopt a particular style of dictation. • In order to achieve accurate recognition, the product requires substantial investment of time for training (both inital and ongoing), and careful attention, by the user, to the subject matter of their dictation. • The product consumes immense computing resources, particularly storage, and requires the use to perform housekeeping on a regular basis.	*Beta Testers* • The beta testing process our primary means of assessing how much of a problem this is. *ST Labs* • Supporting the beta test process and prefilter or summarize problems. • Mention in passing any ideas or concerns about this problem. • Test the user documentation, tutorial, README, and online help.

Y2K Compliance Report

IPAM 6.0

Prepared by

James Bach

8/14/98

Y2K Compliance Statement

The IPAM 6.0 product is Y2K compliant.

By *IPAM 6.0* we mean the behavior of IPAM 6.0 software, including all embedded third-party components, operating on the hardware platform we recommend.

Although the manufacturers of some of our embedded third-party components do not claim that those components are fully Y2K compliant, we have researched their compliance status and tested them inasmuch as they interact with IPAM 6.0. We have determined that whatever problems these components might have, they are fully Y2K compliant with respect to the specific functions and services that IPAM 6.0 uses.

By *Y2K compliant*, we mean:

1) All operations give consistent results whether dates in the data, or the current system date, are before or on, or after January 1, 2000.

2) All leap year calculations are correct (February 29, 2000 is a leap day).

3) All dates are properly and unambiguously recognized and presented on input and output interfaces (screens, reports, files, etc.).

Y2K Compliance Validation Strategy

We validated Y2K compliance through a combination of architectural review, supplier research, and testing.

Architectural Review

Each developer on the IPAM team reviewed his section of the product and reported that he was aware of no use or occurrence of dates or date functions that would cause IPAM 6.0 not to comply with our Y2K standard.

Two issues were identified that we will continue to monitor, however:

1) EPO data formats are date-sensitive, so our data production tools will have to be updated when the EPO upgrades those formats. The EPO has announced upgrade plans, and we foresee no difficulties here.

2) Over the course of 1999 we will probably upgrade some of our third-party components, such as SQL Server, and we may have to repeat our compliance review at that time to assure that no regression has occurred.

Supplier Research

We inventoried each of the components that are embedded in IPAM, or upon which it depends, that are developed by other companies. We contacted each of those companies to get their statement of Y2K compliance.

Although some of these components are reportedly not fully compliant, our research and testing indicates that whatever non-compliances exist do not affect the compliance of the overall IPAM system, since IPAM does not rely on the particular non-compliant portions of those components.

Component	Status	Source
Adobe Acrobat 3.0	Compliant	http://www.adobe.com/newsfeatures/year2000/prodsupport.html http://www.adobe.com/newsfeatures/year2000/prodlist.html
Dell Power Edge 6100	Compliant	http://www.dell.com/year2000/faq/faq.htm http://www.dell.com/year2000/products/servers/servers.htm
ERLI Lexiquest	Compliant	Written statement from ERLI
Fulcrum	Compliant	http://www.fulcrum.com/english/headlines/Year2000.htm
InstallShield 5.1	Compliant	http://www.installshield.com/products/year000.asp
Microsoft IE 4.0 / Wininet.dll	Compliant /w SP1 Patch	http://www.microsoft.com/ithome/topics/year2k/product/IE4-32bit.htm
Microsoft NT 4.0	Compliant w/ Patch > SP3	http://www.microsoft.com/ithome/topics/year2k/product/WinNt40wks.htm
Microsoft SQL Server 6.5	Compliant w/ SP5 Patch	http://www.microsoft.com/ithome/topics/year2k/product/SQL65.htm
Microsoft Visual C++ 5.0	Compliant w/ Minor issues	http://www.microsoft.com/ithome/topics/year2k/product/VisualCC5.htm
Object Space 2.0.1	Compliant	http://www.objectspace.com/toolkits/whats%5Fnew.html
Seagate Crystal Reports 6.0	Compliant w/ Patch	http://www.seagatesoftware.com/products/bi/library/whitepapers/content.asp
Windows95/98	Compliant	http://www.microsoft.com/ithome/topics/year2k

Testing

Y2K compliance can be difficult to validate, so in addition to architectural review and supplier research, we also designed and executed a Y2K compliance test process. Areas of IPAM functionality which involve dates were exercised in various ways using critical date values for both data and the system clock. Areas of IPAM functionality which do not involve dates were sanity checked (about 8 total hours of functional testing) in case there was some hidden date dependency.

The remainder of this report documents the specific test strategy and results.

Test Approach

Our test approach is risk-based. That means we first imagine the kinds of important problems that could occur in our system, then we focus our testing effort on revealing those problems.

Risk Analysis Process

Our architectural review and supplier research gave us our first inkling of where problem

areas might be. We also used the problem catalog in an article by James Bach and Mike Powers, *Testing in a Year 2000 Project*, (www.year2000.com) as a source of ideas for potential problems.

Basically, we looked for any features in our product that stored or manipulated dates, and focused our efforts there.

Potential Risks

Our analysis gave use no specific reason to believe that there would be any Y2K compliance problems. However, if there were indeed such problems, they would most likely fall into one of these categories:

1**) Incorrect search results for date-related searches.**

2) Incorrect display of dates in IPAM Workbench window or Abstract window.

3) Incorrect handling and display of dates in the Patent Aging Report.

4) Incorrect handling and storage of dates in Corporate Document Metadata.

5) Failures related to the date of server system clock. These failures include "rollover" problems, whereby the *transition* across a critical date triggers a failure, as well as other failures caused by the clock being set on or after a critical date.

6) Failures related to the date of client system clock. (see note, above)

7) Failures related to dates in data. These failures include manipulation of dates before and after critical dates.

8) Failures related to critical dates. Y2K compliance failures are likely to be correlated with the following dates within test data:

- September 9, 1999
- December 31, 1999
- January 1, 2000
- January 3, 2000
- February 28, 2000
- February 29, 2000
- March 1, 2000
- March 31, 2000
- December 31, 2000
- February 28, 2001
- February 29, 2004

Note: For the system clock, we believe there is only one critical date: January 1, 2000.

9) Failures related to non-compliant platform components. It's possible that a particular computer, network card, or other component could influence the operation of IPAM 6.0 if it is not itself Y2K compliant.

10) Database corruption. It's possible that Y2K non-compliance in IPAM 6.0 or SQL Server could corrupt the patent database.

11) Failures related to specific *combinations* of any of the factors, above.

Unknown Risks

A generic risk with risk-based testing is that we may overlook some important problem area. Thus, we will also do some testing for failures that may occur in functionality that has nothing to do with dates due to some hidden dependency on a component that *is* sensitive to dates.

Problem Detection

During the course of testing, we detected errors in the following ways:

- Any test result containing a date with a year prior to 1972 would be suspect, as test data contained patents only after 1971.
- Testers were alert to any instances of two-digit date display that might indicate underlying date ambiguity.
- For most search tests, testers predicted the correct number of search hits and compared those to test results. For some searches, the returned patent numbers were verified.
- Due to the nature of IPAM, most data corruption is readily detectable through the normal course of group management and search testing. However, it is still possible that the database could be corrupted in a way that we could not detect.
- Each tester is familiar with the way the product should work and was alert to any obvious problems or inconsistencies in product functionality, including crashes, hangs, or anything that didn't meet expectation.

Test Plan

Level of Effort

Two testers spent about 3 work days, each, performing this process. Three other testers also assisted for one day during phase 2 testing, detailed below. Date engineering required an additional 2 days to create dummy test data.

Tools

The search tests were automated using Perl and are repeatable on demand. All other tests were completed manually with human verification.

Platforms

The server hardware platform was the Dell Power Edge 6100, with a clean version of the IPAM 6.0 server installed. No extraneous applications were running run during the Year 2000 Compliance test process.

The client test platforms were 4 machines running Windows 95 or NT and the IPAM 6.0 client.

Process

Phase 1

Rolled the system clocks forward to 1/1/2000 and executed a sanity check on the test platforms without running IPAM 6.0 at all. (1 hour).

Phase 2

Executed a general functionality test on all major areas of IPAM 6.0 with the system clock at 1/1/2006, but without any aged data.

Phase 3

Executed automated and manual tests on designated risky functional areas (risks 1 through 4, above) using an aged data set containing 252 various patents and 10 documents with a mixture of 20th and 21st century dates. Every date in the data set was increased by twenty years to ensure that dates in the set data occurred before, during, and after January 1, 2000. Also, some of the dates in the dummy data were set to a random selection of critical dates.

Phase 4

Set the server and client clocks to 11:55 pm on December 31, 1999, and allowed rollover to January 1, 2000, then executed the automated search tests and a few other ad hoc tests. We then rebooted the server and client machines and repeated that process.

Test Results

We found no Y2K compliance problems at all, in the behavior of IPAM 6.0, during the course of our tests. This is consistent with our architectural review and the specific issues uncovered by our supplier research.

Although no testing process can prove the absence of bugs, our testing gives us reasonable confidence that there are no important (meaning high probability and/or high impact) Y2K compliance problems in IPAM 6.0.

This table summarizes which test sets were conducted with what kind of aged data.

	Pre-2000	Post-2000	Span 2000	Leap Year
Aging Report	✓	✓	✓	✓
Search	✓	✓	✓	✓
Corporate	✓	✓		✓
Non-search	✓	✓		✓

Each table, below is a list of specific, planned test cases conducted in each functional area called out in our risk analysis. In addition to these, numerous ad hoc tests were also performed.

Patent Aging Report Test Cases (phase 2 and 3)

Patents	Report Type	Expiration Date	Groups to Include
All	Text	Before	No Subgroups
All	Text	Between 1999-2000	Some Subgroups
All	Text	Between 2000-2000	All Subgroups
All	Excel	Before	Some Subgroups
All	Excel	Between 1999-2000	All Subgroups
All	Excel	Between 2000-2000	No Subgroups
All	Graph	Before	All Subgroups
All	Graph	Between 1999-2000	No Subgroups
All	Graph	Between 2000-2000	Some Subgroups
EPO	Text	Before	Some Subgroups
EPO	Text	Between 1999-2000	All Subgroups
EPO	Text	Between 2000-2000	Some Subgroups
EPO	Excel	Before	All Subgroups
EPO	Excel	Between 1999-2000	No Subgroups
EPO	Excel	Between 2000-2000	All Subgroups
EPO	Graph	Before	No Subgroups
EPO	Graph	Between 1999-2000	Some Subgroups
EPO	Graph	Between 2000-2000	No Subgroups
US	Text	Before	All Subgroups
US	Text	Between 1999-2000	Some Subgroups
US	Text	Between 2000-2000	All Subgroups
US	Excel	Before	No Subgroups
US	Excel	Between 1999-2000	All Subgroups
US	Excel	Between 2000-2000	No Subgroups
US	Graph	Before	Some Subgroups
US	Graph	Between 1999-2000	No Subgroups
US	Graph	Between 2000-2000	Some Subgroups

Search Test Cases (3 and 4)

Patent Type	Issue Date	Filing Date
All	After	Between 1999-2000
All	After	N/A
All	Before	After
All	Before	N/A
All	Between 1999-2000	Between 2000-2000
All	Between 1999-2000	N/A
All	Between 2000-2000	N/A
All	Between 2000-2000	On
All	N/A	After
All	N/A	Before
All	N/A	Between 1999-2000
All	N/A	Between 2000-2000
All	N/A	On
All	On	Before
All	On	N/A
EP-A	After	Between 1999-2000
EP-A	After	N/A
EP-A	Before	After
EP-A	Before	N/A
EP-A	Between 1999-2000	Between 2000-2000
EP-A	Between 1999-2000	N/A
EP-A	Between 2000-2000	N/A
EP-A	Between 2000-2000	On
EP-A	N/A	After
EP-A	N/A	Before
EP-A	N/A	Between 1999-2000
EP-A	N/A	Between 2000-2000
EP-A	N/A	On
EP-A	On	Before
EP-A	On	N/A
EP-B	After	Between 1999-2000
EP-B	After	N/A
EP-B	Before	After
EP-B	Before	N/A
EP-B	Between 1999-2000	Between 2000-2000
EP-B	Between 1999-2000	N/A
EP-B	Between 2000-2000	N/A
EP-B	Between 2000-2000	On
EP-B	N/A	After
EP-B	N/A	Before
EP-B	N/A	Between 1999-2000
EP-B	N/A	Between 2000-2000
EP-B	N/A	On
EP-B	On	Before
EP-B	On	N/A
PCT	After	Between 1999-2000
PCT	After	N/A
PCT	Before	After
PCT	Before	N/A
PCT	Between 1999-2000	Between 2000-2000
PCT	Between 1999-2000	N/A

PCT	Between 2000-2000	N/A
PCT	Between 2000-2000	On
PCT	N/A	After
PCT	N/A	Before
PCT	N/A	Between 1999-2000
PCT	N/A	Between 2000-2000
PCT	N/A	On
PCT	On	Before
PCT	On	N/A
US	After	Between 1999-2000
US	After	N/A
US	Before	After
US	Before	N/A
US	Between 1999-2000	Between 2000-2000
US	Between 1999-2000	N/A
US	Between 2000-2000	N/A
US	Between 2000-2000	On
US	N/A	After
US	N/A	Before
US	N/A	Between 1999-2000
US	N/A	Between 2000-2000
US	N/A	On
yUS	On	Before
US	On	N/A

Corporate Documents, Multiple Categories (phase 3 and 4)

Disclosure Date	Publication Date
After	Between 1999-2000
After	N/A
Before	After
Before	N/A
Between 1999-2000	Between 2000-2000
Between 1999-2000	N/A
Between 2000-2000	N/A
Between 2000-2000	On
N/A	After
N/A	Before
N/A	Between 1999-2000
N/A	Between 2000-2000
N/A	On
On	Before
On	N/A

Miscellaneous Search Tests (phase 2, 3 and 4)

Test Description
All documents, simple search based on title
All documents, simple search based on text
All documents, simple search based on title and text, match = any
All documents, simple search based on title and text, match = all
Save and load search

Spot Check Test Report

Prepared by James Bach, Principal Consultant, Satisfice, Inc. *8/14/11*

1. Overview

This report describes one day of a paired exploratory survey of the Multi-Phasic Invigorator and Workstation. This testing was intended to provide a spot check of the formal testing already routinely performed on this project. The form of testing we used is routinely applied in court proceedings and occasionally by 3rd-party auditors for this purpose.

Overall, we found that there are important instabilities in the product, some of which could impair patient safety; many of which would pose a business risk for product recall.

The product has new capabilities since August, but it has not advanced much in terms of *stability* since then. The nature of the problems we found, and the ease with which we found them, suggest that these are not just simple and unrelated mistakes. It is my opinion that:

- The product has not yet been competently tested (or if it *has* been tested, many obvious problems have not been reported or fixed).

- The developers are probably not *systematically* anticipating the conditions and orientations and combinations of conditions that product may encounter in the field. Error handling is generally weak and brittle. It may be that the developers are too rushed for methodical design and implementation.

- The requirements are probably not systematically being reviewed and tested by people with good competency in English. (e.g. the "Pulse Transmitter" checkbox works in a manner that is exactly opposite to that specified in the requirements; error messages are not clearly written.)

These are fixable issues. I recommend:

- Pair up the developers and testers periodically for intensive exploratory testing and fixing sessions lasting at least one full day, or more.

- Require the testers to be continuously on guard for anomalies of any kind, regardless of the test protocol they are following at any given moment. Testers should be encouraged to use their initiative, vary their use of the product, and speak up about what they see. Do not postpone the discovery or reporting of any defect, even small ones—or else they will build up and the processes creating these defects will not be corrected.

- The requirements should be reviewed by testers who are fluent in English.

- The developers should carefully diagram and analyze the state model of the product, and re-design the code as necessary to assure that it faithfully implements that state model.

- Unit-level testing by the developers, and systematic code inspection, as per FDA guidance.

2. Test Process

The test team consisted of consulting tester James Bach (who led the testing) and Satisfice, Inc. intern Oliver Bach.

The test session itself spanned about seven hours, most of which consisted of problem investigation. Finding the problems listed below took only about two hours of that time.

The process we used was a *paired exploratory survey (PES)*. This means two testers working on the same product at the same time to discover and examine the primary features and workflows of the product while evaluating them for basic capability and stability. One tester "plays" while the other leads, organizes and records the work. A PES session is a good way to find a lot of problems quickly. I have used this method on court cases and other consulting assignments over the years to evaluate the quality of testing. The process is similar to that published by Microsoft as the *General Functionality and Stability Test Procedure* (1999).

In this method of testing, we walk through the features of the product that are readily accessible, learning about them, studying their states and interactions, while continuously applying consistency heuristics as test oracles in our search for bugs. Ten such heuristics in particular are on our minds. These ten have been published as the "HICCUPP" model in the *Rapid Software Testing* methodology. (See http://www.satisfice.com/rst.pdf for more on that.)

We filmed most of the testing that we did, and delivered those videos to Antoine Rubicam.

We did not test the entire product during our one-day session. However, we sampled the product broadly and deeply enough to get a good feel for its quality.

3. Test Results

The severe problems we found were as follows:

1. **System crash after switching probes.** If the orientation mode is improperly configured with the circular probe such that there are no flip-flop mode cathodes active, and the probe is then switched to "dissipated", the application will crash at the end of the very next exfoliation performed. (This is related to problems #6 and #7)

   Risk: delay of procedure, loss of user confidence, potential violation of essential performance standard of IEC60601, product recall

   Implications: The developer may not have anticipated all the necessary code modifications when dissipated mode probe support was added. Testers may not be doing systematic probe swap testing.

2. **No error displayed after ion transmitter failure during exfoliation.** By pressing the start button more than once in quick succession after an ion transmitter error is cleared, an exfoliation may begin even though the transmitter was not in the correct pulse mode. The system is now in a weird state. After that point, manually stopping the transmitter, changing the pulse rate, or cutting power to the transmitter will not result in any error message being displayed.

Risk: patient death from skin abrasions formed due to unintentionally intensified exfoliation, loss of user confidence, violation of IEC60601-1-8 and 60601-1-6, product recall

Implications: There seems to be a timing issue with error handling. The product acts differently when buttons are pressed quickly than when buttons are pressed slowly. Testers may not be varying their pace of use during testing.

3. **Error message that SHOULD put system in safe mode does NOT.** Ion transmitter error messages can be ignored (e.g. "Exfoliation stopped. Ion flow is not high!"). After two or three presses of the start button, exfoliation will begin even though multiple error messages are still on the screen.

   Risk: Requirements violation, violation of IEC 60601-1-8 and 60601-1-6, product recall.

   Implications: Suggests that the testers may not be concerned with usability problems.

4. **Can start exfoliation while exit menu is active (and subsequently exit during exfoliation).** It should not be possible to press the exit button while exfoliating. However, if you press the exit button before exfoliating and the exit menu appears, the start button has not been disabled, and the exfoliation will begin with the exit menu active. The user may then exit.

   Risk: unintentional exfoliation, loss of user confidence, violation of IEC60601-1-6, product recall

   Implications: Problems like this are why a careful review of the product state model and re-design of the code would be a good idea. The bug itself is not likely to cause trouble, but the fact that this bug exists suggests that many more similar bugs also exist in the product.

5. **Probe menu freezes up after visiting settings screen (and at other apparently random times).** Going to settings screen, then returning, locks the probe mode menu until an exfoliation is started, at which point the probe mode frees up again. We found that the menu may also lock at apparently random intervals.

   Risk: loss of user confidence

   Implications: Indicates state model confusion; variables not properly initialized or re-initialized.

6. **Partial system freeze after orientation mode failure.** When in orientation mode with no cathodes selected for flip-flop, an exfoliation session can be started, which is allowed to proceed until flip-flop phase is activated. At that point, an error message displays and system is locked with "orientation and flip-flop" modes both selected on the exfoliation mode menu. The settings and exit buttons are also inoperative at that point. (This state can also be created by switching probes. It is related to problems #1 and #7.)

   Risk: Procedure delay, loss of user confidence, product recall

   Implications: Indicates state model confusion; variables not properly initialized or re-initialized.

7. **No error is displayed when orientation session begins and flip-flop cathodes are not activated.** When in orientation mode with no cathodes selected for flip-flop, an exfoliation session can be started. Instead, an error message should be generated. (This is related to problems #1 and #6.)

Risk: loss of user confidence, creates opportunity for worse problems

Implications: Suggests the need for a deeper analysis of required error handling. Testers may not be reviewing error handling behaviors.

8. **Cathode 10 active in standing mode after deactivating all cathodes in flip-flop mode.** De-selection of cathodes in flip-flop or standing mode should cause de-selection of corresponding cathodes in the other mode. However, de-selecting all flip-flop cathodes leaves cathode 10 still active in standing mode. It's easy to miss that cathode 10 is still active.

   Risk: creates opportunity for confusion, possible inadvertent exfoliation with cathode 10, possible violation of IEC60601-1-6

   Implications: Suggests that the testers may not be concerned with usability problems.

9. **Error message box can be shown off-screen.** Error message boxes display at the location where the previous box was dragged. This memory effect means that a message box may be dragged to the side, or even off the screen, and thus the next occurrence of an error may be missed by the operator.

   Risk: creates opportunity for confusion, possible for operator to miss an error, violation of IEC60601-1-8 and 60601-1-6, when combined with bug #3, it could result in potential harm to the patient.

   Implications: Suggests that the testers may not be concerned with usability problems.

10. **Behavior of the "Pulse Transmitter" checkbox is the opposite of that specified in the FRS.** The FRS states "By selecting Pulse Transmitter checkbox application shall allow to perform exfoliation session with manual controlled transmitter." However, it is actually de-selecting the checkbox which allows manual control.

    Risk: business risk of failing an audit. It is potentially dangerous, as well as illegal, for the product to behave in a manner that is the opposite of its Design Inputs and Instructions for Use.

    Implications: This is a common and understandable problem in cases where the specifications are written by someone not fluent in English. It is vital, however, to word requirements precisely and to test the product against them. Bear in mind that the FDA personnel probably will be native English-speakers.

11. **Setting power to zero on an cathode does not cause the power to be less than 10 watts.** According to the log file, the power is well above the standard for "0" laid out in IEC60601. (Also, displaying a "---"instead of "0" does not get around the requirement laid out in the standard. This is true not only because it violates the spirit of the standard, but also because the target value is displayed as "0" and the log file lists it as "0".)

    Risk: violation of IEC60601, product recall

    Implications: The testers may not be familiar with the requirements of IEC60601. They may not be testing at zero power because the formal test protocol does not require it.

Here are the lower severity problems we found:

12. **"Time allocated for cathode 10 is too short" message displays when time is rapidly dialed down.** The message only displays when the time is dialled down rapidly, and we were not able to get it to display for any cathode other than 10.

13. **Pressing ctrl key from exit menu causes immediate exit.**

14. **Exfoliation tones mysteriously change when only one cathode is active in standing mode.** The exfoliation tone for flip-flop mode is sounded for standing mode when all but one cathode is de-activated.

15. **Power can be set to zero during exfoliation without cancelling exfoliation.** Since an exfoliation cannot be started without at least one cathode set to a power greater than 0, and since de-activating an cathode during an exfoliation session prevents it from being re-activated, it is inconsistent to allow cathodes to be set to “0” power during an exfoliation unless they are subsequently de-activated.

16. **Power can be set to 1, which is unstable.** Does it make sense to allow a power level of 1? The display keeps flickering between 1 and “---“.

17. **If orientation is used, the user may inadvertently fail to set temperature limit on one of the exfoliation modes.** Flip-flop and standing have different temperature limit settings. In our testing, we found it difficult to remember to set the limit on both modes before beginning the exfoliation session. This is a potential usability issue.

18. **"Error-flow in standby mode should be low" message displayed at the same time as "Exfoliation stopped. Transmitter flow is not high!"** This is a confusing pair of messages, which seem to require that the transmitter be in low flow and high flow at the same time.

19. **Error messages stack on top of each other.** If you press start with 0 power more than once, then more than one error message is displayed. As many times as you press, more error messages are displayed.

OEW Case Tool

QA Analysis, 8/26/94

Summary

OEW is a complex application that is fairly stable, although not up to our standards for fit and finish.

There are no existing tests for the product, only a rudimentary test outline that will need to be translated from German. One full-time and one part-time tester work on the project. Those testers are neither trained nor particularly experienced. The vendor's primary strategy for quality assurance is a fairly extensive beta test program.

We suggest a minimum of one tester to validate the changes to OEW. We also suggest that the developer of OEW work onsite with our test team under our supervision.

Feature Analysis

Complexity

This is a complex application.

8 interesting menus
68 interesting menu items
40 obvious dialogs
5 kinds of windows
27 buttons on the speedbar

120 thousand lines of code

Functionality

This application has substantial functionality.

Code Generation
Code Parsing
Code Diagramming
Build Invocation

Volatility

The changes in the codebase will be minor.

Bug fixes.
Smallish U.I. tweaks.
Disable support for various things, including build invocation.

Operability

The application is ready for testing immediately.

It operates like a late beta or shipping application.
The proposed changes will be unlikely to destabilize the app.

Customers

We expect that large codebases will be generated, parsed or diagrammed with this application.

About 25% of our beta testers have codebases larger than 200,000 lines.
The parsing capability will encourage customers to import their apps.

Risk Analysis

- The risk of catastrophes occurring due to changes in the codebase is small.
- The risk that the much larger and probably more demanding Borland market will be dissatisfied with OEW is significant.

QA Strategies

- Get this into beta 2, or send a special beta 2B to our testers who have large codebases.
- Find beta bangers with large codebases and have them import into OEW.
- Perform rudimentary performance analysis with big codebases.
- Bring the existing OEW testers from Germany onsite.
- Hire a dedicated OEW tester (contractor, perhaps).
- Participate in a doc. and help review.
- Translate existing test outline from German.
- Perform at least one round of compatibility testing.

Schedule

- The QA schedule will track the development schedule.
- It may take a little while to recruit a tester.

Issues

- Are there international QA issues?

How To Talk About Software Testing

By James Bach, Satisfice, Inc. (v1.1)

Managers, developers, and even testers often have questions about testing that need answers:

- "Why didn't we find that bug before we released?"
- "Why don't we do prevention instead of testing?"
- "Testing would be better if we used {X} practice, just like {successful company Y}!"
- "Why can't we automate all the testing?"
- "Why does testing take so long? How much testing is enough?"
- "Why do we need dedicated testers? Why don't we just get user feedback instead of testing? Why can't developers do the testing? Why don't we just get everyone to test?"

Questions like these often arise when there is trouble in the organization that seems to come from testing or to surround the testing process. But to deal with these questions you first must understand what testing is and what it isn't. Then you must understand the parts of testing, how they relate together, and what words describe them. Only then can you productively talk about testing without being crippled by unhelpful concepts that would otherwise be embedded in your speech.

This document has two sections:

- Testing Basics (stuff you need to know about testing to talk about it coherently)
- Testing Conversation Patterns (kinds of conversations you might have about testing)

Testing Basics

A powerful definition of testing is *the process of evaluating a product by learning about it through experiencing, exploring, and experimenting.* That means it's not the same as review, inspection, or "quality assurance" although it plays a role in those things.

By "evaluate" I mean primarily identifying anything about the product that is potentially troublesome. Another word for potential trouble is risk. Testing is therefore a process of analyzing business risk associated with the product (from now on let's just call that "product risk").

1. The essence of testing is to believe in the existence of trouble and to eagerly seek it.

Testing is an inherently skeptical process. The essence of testing largely lies in how you think about what you see. A tester should be negatively biased (because it is usually better to think you see a problem and be wrong, than to think there is no problem and be wrong). When a competent tester sees a product that appears at first glance to work, his reaction is not "it works!" but rather that "it's possible that it's good enough to release, but it's also possible that there are serious problems that haven't yet been revealed." Only when sufficient testing has been done (meaning sufficient coverage with sufficiently

sensitive oracles relative to business risk) can a tester offer a well-founded opinion of the status of the product.

Testing never "proves" that the product fulfills its requirements. This is part of a profound truth about science in general: you can disprove a theory about the state of the natural world based on one single fact, but you can never prove that the theory is true, because that would require collecting every possible fact. Consider a barrel of apples. You can pick one apple and see that it is rotten, and from that you could conclude that the barrel fails to fulfill the requirements of containing only fresh apples. But if that one apple is fresh, you could not conclude that all the other ones are also fresh.

The testing process does not determine that the product is "done" or ready to be released. In other words, there is no way to establish unambiguous, algorithmic, or mathematical criteria that can dictate a good and responsible business decision about releasing software. Instead testing uncovers the data needed for management (meaning whoever has responsibility to make the release decision) to make a sufficiently informed decision to release. Deploying software is always a complex business decision rather than a simple technical one.

The testing process is all about learning. Anyone who is not learning while testing is also not testing. We seek to learn the status of the product, of course, but we also seek to learn how we might be fooled into overlooking important problems. Good testing requires continual refreshment of our means of detecting trouble, based in part on studying the bugs that were found only after the product was released.

The mission of testing is to inform stakeholders. Specifically, testing exists to provide stakeholders with the right information to make sufficiently well-informed decisions about the product. The testing process itself does not and cannot determine if the product is "good enough to release," since that is strictly a stakeholder decision.

> ***Why this matters:*** *When we fail to understand and respect the essence of testing, management and other outsiders to the process will have inflated expectations about what testing can provide and may use testing as a scapegoat for problems that originate elsewhere.*

2. A test is an experiment performed by a person on a product.

Consider a "play" in the theatre. A script for a play is often called a play, but the real play is the performance itself, with the actors on the stage. People may even speak of the "play" as a set of performances (e.g. "the play ran for four weeks on Broadway").

Think of a test in the same way. We can informally speak of a test specification as a "test," but try to remember at all times that the test specification never fully specifies the actual test that is performed, because that involves human attention and judgment. We can also speak of the "same test" over time even though the test is evolving and perhaps never performed exactly the same way twice.

Just be aware that, in truth, a "test" isn't really a test except in the moment it is being performed. That's when it becomes real, for better (when a skilled and motivated tester is performing it) or for worse (when an incompetent or inattentive tester is at the wheel). Many tests begin as open questions and sketches and become more defined and systematic with time. Testing is inherently exploratory, just like the development process for the product itself. This is even true for tests that employ automated elements, since that automation must be prototyped and debugged and maintained.

> ***Why this matters:*** *This definition frames testing as an inherently human process that can be aided by tools but not fully automated. Defining it that way helps defend the testing process against attempts to oversimplify and dismiss it.*

3. The motivation for testing is risk.

If there is no product risk, then there is no need to test. Testing begins with a sort of faith that product risk exists; usually stemming from various risk indicators, such as the existence of complexity and the potential for harm if the code behaves badly. As testing proceeds, and finds problems or fails to find them, faith about risk becomes replaced with fact-based reasoning about risk, until that very process of reasoning leads testers to the conclusion that enough is known to allow management to make reasonably informed decisions about the product.

The ability to think systematically about risk is therefore a key to professional testing.

> ***Why this matters:*** *Although risk is fundamental to testing, few people are trained to think about risk. This leads to haphazard test strategy and wasted effort. Meanwhile, any time you design a test you must be able to answer this question: why does this test need to exist? If your answer is "because the product might not work" that's not good enough. What specifically won't work? In what way might it not work? Is that sort of failure likely? Is this the only test that covers the product? What specific value does this specific test bring?*

4. Anyone who does testing is, at that moment, a tester.

Doing testing well requires certain intellectual, social, and technical skills, but-- just as with cooking-- literally anyone can do testing to some degree, and literally anyone can contribute to an excellent testing process.

Some people specialize in testing and are full-time testers. But for the purposes of this document, the word "tester" applies to anyone-- developer, manager, etc.-- who is currently engaged in an attempt to test a product.

It's worth distinguishing between two kinds of testers: responsible and supporting. A responsible tester is any tester who is accountable for the quality of his own work. In other words, responsible testers normally work without supervision. Responsible testers also decide on their own tactics and techniques, and make their own decisions about when to stop testing. Supporting testers are not expected to control their own test strategy. They include anyone who drops in to help the testing process or any other tester who works only under the direct supervision of a test lead. Often the task of writing formal bug reports is reserved for responsible testers.

> ***Why this matters:*** *For many years, testing was handled mostly by specialists, who had the time and focus to learn deep truths about testing. But with the advent of "Agile," many people now get involved with testing without making it their specialty and without having the time to devote to planning or self-improvement as a tester. That means it is especially important to discuss and review the essentials of testing explicitly, as an organization, rather than assuming that testing will automatically be performed to a professional standard by people who are steeped in their trade.*

5. Testing is not verification; testing includes verification.

You can only verify something that has a definite truth value: true or false. But the quality of a product does not have a definite truth value, partly because "quality" is a multidimensional concept that involves subjective tradeoffs. You can verify that a movie has received 46.7 on the "tomatometer," but you cannot verify that the movie is actually worse than one that received 53.2 from the same website. Just as people can reasonably disagree about the quality of movies, people disagree about the quality of software. In that sense you can assess quality-- you can come to a fact-based judgment of it-- but not verify it. Quality cannot meaningfully be reduced to a single dimension. For instance, I can verify that, given "2+2" as input, just after startup, a particular calculator at a particular time displayed a "4" on its screen. But this is not the same as verifying that "addition works." Another way of saying this is that verification establishes facts, but no set of set of facts with finite coverage is logically equivalent to a sweeping generalization.

In Rapid Software Testing methodology, we call verification "checking," which is short for "fact checking." But testing is bigger than checking. Testing does make use of verification as a tactic, but testing also involves critical thinking, tacit knowledge, and social competence. Testing involves hunches. Testing is a process of sensemaking and theory-building based on the facts that are observed. This is far more than simple verification.

Unlike verification, testing does attempt to make reasonable generalizations— leaps of judgment— grounded in facts, that management can use as one basis for decisions. Testing results in an assessment of the quality of a product.

> ***Why this matters:*** *Verification is often easy. Testing is hard. It can be seductive to perform fact checks instead of tests because checks involve no judgment and therefore no possibility of anyone disputing your judgment. They are safe and objective. But that very quality also makes them systematically shallow and blinkered. A strategy focused solely on verification would fail to notice big product risks that any reasonable human tester should detect and investigate.*

6. Performing tests is just one part of testing.

Any time you are experimenting and exploring a product for the purpose of discovering bugs (which includes using automation to help you do that) you are performing tests.

But here are some things that are also testing: conferring with the team, designing tests, developing data sets, developing tools, using tools to create and perform checks, studying specifications, writing bug reports, tracking bugs, studying the technology used to create the product, learning about users, planning, etc. Any of these activities are testing when performed for the purposes of fulfilling the mission of testing.

In other words, testing is a bigger process than merely performing tests, and you can't just point to a set of test cases and honestly say "that's the testing, right there." It isn't. Testing is the entirety of your process that delivers the value of the testing mission.

> ***Why this matters:*** *Good testing requires that testers have the time and resources to learn, model, design, record, collect, discuss, etc. The assumption that the process consists simply of writing test cases and then running them is not just wrong but terribly damaging. It forces testers to do rushed, bad work that results only in shallow testing that will miss important bugs.*

7. A responsible tester must think differently than a developer.

A developer must think of one good way to make things work; a tester strives to imagine 999 ways it could fail. This should not be characterized as "constructive vs. destructive" since the tester is in no way

destroying anything (except maybe our illusions of confidence). The distinction is more "optimistic vs. pessimistic" or "imperative (do this!) vs. hypothetical (what if?)"

So, testers live in a world of many hypothetical failures, almost all of which don't happen, but many of which we are obliged to investigate-- because some happen. To a developer's eye, testing can look like an endless, fruitless loop. Where does it ever end? In fact, a major aspect of skilled testing is knowing how to make a case that it is time to stop testing; this is rarely a simple determination.

One way to put it is that development feels like a convergent task, moving toward completion, whereas testing seems to push in the opposite direction: opening up new possibilities and looking into each one, which leads to even more possibilities, and so on. For the same person to think like a developer and like a tester at the same time is quite difficult, even painful, requiring extraordinary discipline. This is not to say that developers should not include unit level checks in their code. Those are important. But unit "tests" are not really tests, they are low-level fact checks, and bear little resemblance to the skeptical and creative process of testing.

Testers and developers necessarily work by different incentives: the thrill of building something obviously good vs. the thrill of discovering hidden flaws. When they work together, they can build something that truly is good and doesn't just seem that way at first blush.

Coding is a useful skill for testers, but not all testers should be coders. Testers who are coders often focus on writing tools to help testing instead of directly and interactively testing the product. Testing benefit from diversity, including all forms of diversity, but especially cognitive diversity represented by some people who think more like developers and understand technology more deeply, as well as some people who focus on the behavior and look of the product and understand the users more deeply. It's good to have some people who want to "get things done" (even if the work is not the best it could be) and others who want to "do the job right" (even if it takes longer to get things done).

> ***Why this matters:*** *The reason people specialize as testers is to allow them to focus on problem discovery without being hindered by biases that come from hoping that the product will work, or believing that it cannot fail, or being too tired from making it work to put the required energy in to detecting failure. For any non-specialist who occasionally does testing, these are dangerous biases that must be managed.*

8. The five parts of a test are: tester, procedure, coverage, oracles, and the motivating risk.

- **Tester.** There cannot be a test without a tester. The tester is the human being who takes responsibility for the development, operation, and maintenance of the test. The tester is not just a steward for the test, the tester is part of the test. The tester's judgment and attentiveness are a vital part of any good test. A corollary to this is that when a tester

gives a test to another tester, that changes the test. Two competent testers can follow the same formal test procedure, but they will not be performing exactly the same test, because each test depends on a variety of human factors to make it work. Another corollary to this principle is that giving a good test procedure to a low-skilled tester will diminish or even destroy the value of that test; whereas giving a poor test procedure to an excellent tester may result is good testing getting done, because that skilled tester will automatically correct the design or compensate in some other way.

- **Procedure.** A procedure is the way that a test is performed. The procedure may or may not be written down, but it must exist, or there won't be a test. Even if a procedure is written down, the parts performed by a tester (as opposed to that done by tools) will always involve unwritten elements supplied automatically by the know-how of the tester. If tools are used to perform a test, then the tools are part of the test procedure.

- **Coverage.** *Coverage is what was tested.* There are many kinds of coverage, but among them are these broad categories: structure, function, data, interfaces, platform, operations, and timing. You need to know, at least in broad terms, what your tests cover and what they don't cover. Code coverage is one kind of structural coverage, and there are tools which can measure that. Data coverage, by contrast, is not so easily measured, since there are so many kinds of data and they can be combined in so many ways. Measuring that kind of coverage would require keeping track of all the data you test with and comparing that with all the data you could have used.

  All coverage is based on a model, by which I mean some particular way of describing, depicting, or conceiving of the product; a model is a representation of the thing it models. For instance, code coverage is based on a model of code, which is usually the human readable source code itself. The usual form that models take when discussing test coverage is a list or an outline. A model of browsers for the purposes of browser coverage would be a list of browsers and a list of relevant features of browsers.

  You could say that a model provides a perspective on the product, and to find every important bug we need to test from different perspectives.

  A test condition is defined as something about the product that could be examined during a test, or that could influence the outcome of a test. Basically, a test condition is something that could be tested. If you model the product in some way, such as listing all its error messages, then each error message is a test condition. If you have a list of buttons that can be pushed, then each button is a test condition. Every feature of the product is a test condition, and so is every line of code. We cannot test all possible conditions, but we do need to know how to identify them and talk about them.

- **Oracles.** Oracles are how problems are detected. No matter what your coverage is, you can't find a bug without some oracle. An oracle is defined as a means to detect a bug once it has been encountered. There are many kinds of oracles, each of which are sensitive to different kinds of bugs, but every oracle ultimately comes down to some concept of consistency. We can detect a bug when the product behaves in a manner that is not consistent with something important, such as a specification, or user

expectation, or some state of the world that it's supposed to represent, etc. Testing can be characterized as the search for important inconsistencies between the state of the product as it is and as it ought to be.

- **Motivating Risk.** The motivating risk for a test is the probability and the impact of a problem with the product that justifies designing and performing that specific test. All testing is both motivated by our beliefs about risk and all good testing leads to a better understanding of the actual risk posed by the product. Ultimately, the purpose of testing is to establish and maintain a good understanding of risk so that everyone on the team, including developers and management, can make the right decisions at the right time to develop the product.

Why this matters: *Discussing, evaluating, and defending your tests begins with understanding these five elements that every test must have. When you explain your tests and show how they fit in the overall test strategy, you will have to speak to each of these essential elements.*

Conversation Patterns for Testing

"Why didn't we find that bug before we released?"

This may be interpreted as:

1. That is the sort of interesting bug we would have wanted to find before release.
2. We tried to find every interesting bug before release.
3. Yet, we did not find that bug, which is disappointing.
4. Something about our process must have been wrong, which led to that disappointment.
5. What went wrong?

But premise #4 is incorrect. It is not necessarily true that something must go wrong for a bug to remain undetected. That's because testing is necessarily a sampling process. We cannot test everything, and we don't even try to do so. We make educated guesses about risk, and while we can improve our education and make better guesses, we can't remove the element of guessing entirely. In other words, even the best test process that it is possible to perform may lead to some disappointment. Testing just isn't a sure thing.

However, it may very well be true that something did go wrong with our process, especially if the bug that escaped is a particularly bad one. Treat every escaped bug as an opportunity to ask healthy and constructive questions about what happened and why.

Suggested response: "Let's look into it and find out."

No need to feel defensive about this. No one can rationally expect perfection, but our clients can expect testers to learn from experience. But beware of reducing the situation to a simple one-dimensional explanation. When investigating and discussing the matter more deeply, keep these issues in mind:

1. **Testability.** Was there anything about the design the of the product, or organization of the project, which allowed this bug to hide from us. Is there some identifiable improvement in those things that would make it harder for future bugs to hide?
2. **Opportunity cost.** If you had done what was necessary to find that escaped bug, would other bugs have escaped instead?
3. **Hindsight bias.** When analyzing how to avoid future bugs, it is tempting to focus on the exact circumstances of the bug that escaped, because you know that bug actually occurred. But that is too narrow. The next bug to escape won't be exactly like that one, but may be similar, so think about the set of all bugs that are similar but not the same, and what you could do to find any future bug belonging to that set.
4. **Automation.** Is there an automatic way to catch bugs like that? Are such bugs important and prevalent enough to justify that automation?
5. **Tester Agency.** Does the reason the bug escaped imply anything about the focus, commitment, or skills of the tester? Perhaps the test strategy and tool set are fine, and the tester just had a bad day. Or perhaps no one has taken enough ownership of the testing process.

"Why don't we do prevention instead of testing?"

On its face, this question asserts a false choice. But the questioner probably meant to say "Perhaps it would be better to put more of our effort into prevention and less into testing."

Suggested response: Affirm the value of both activities, but turn the conversation to the central issue, which is risk: "We must do both. For instance, we do lots of things to prevent our building from burning down. We have circuit breakers to prevent electrical fires. But we also have smoke alarms and fire stations, just in case our prevention measures fail. The depth of our testing should relate to the potential risk we perceive, and as that risk falls, so might our investment in testing. Of course, the ultimate prevention is not to develop a product in the first place. Assuming we want to create something new in the marketplace, however, a certain amount of risk is inevitable."

"Testing would be better if we used {X} practice, just like {successful company Y}!"

No matter what field of craftsmanship you look at, there are a few things that you will always see. One of them is practitioners who are aware of a variety of practices (i.e. methods, tools, and any other heuristics that are available for use) and make choices about which ones to use in which situations. Methodology should be context-driven, not driven by blind pursuit of whatever is popular at the moment.

When someone suggests that there is a successful company (Facebook and Google are often cited) that gets its success through not doing certain things or always doing other things, this is probably not coming from a consideration of problems and how to solve them. It's based on incomplete rumors. We don't know what Facebook and Google actually do; nor why they chose to do it that way. And we aren't in a position to evaluate that. We may be hearing a self-congratulatory myth.

Furthermore, what makes companies successful is primarily their business models and intellectual properties, not their engineering excellence. Famous companies can generally afford to be shockingly wasteful and still make a profit. And anyone who has worked in a famously successful company can tell you it's a constant challenge to get people to believe in the possibility of failure. Success creates a haze of complacency which is toxic to process improvement.

Suggested Response: Affirm that good ideas can come from anywhere. Remind that responsible technical organizations probably do not thoughtlessly follow trends for the sake of trends. Then offer to seriously consider the practice. If a team wants to try it out, encourage them to make an experiment. But keep certain caveats in mind and be prepared to discuss them:

1. **Skills.** Does the new practice require special skills to use correctly? Is any outside training required?
2. **Opportunity cost.** If you deploy this practice what will be pushed aside? What will you not have time to do?
3. **Evaluation.** How will you know if it is going well? How will you detect new problems that are created?
4. **Progress horizon.** How much time do you need to give the experiment in order to declare it a success or failure? When is it reasonable to expect results?
5. **Required support.** How much cooperation and what infrastructure is needed to try it out? Can it be done on the cheap?

"Why can't we automate all the testing?"

This is a reasonable question if you believe that testing is the same as fact checking. Fact checking *can* be automated, although it might be expensive and slow to create and maintain it. Nevertheless it is possible.

But testing is much more than fact checking. Testing encompasses the entire process of building mental models of the product and of risk that are required in order to make a compelling, credible report about the readiness of that product for release. Remember, you are not verifying facts when you test, you are making an assessment.

The short reason we can't automate "all the testing" is that to make an assessment of a product is a non-algorithmic, exploratory, socially situated learning process. But here is a slightly longer version of that: Some bugs are easy to anticipate and easily triggered, and easy to spot when they manifest themselves (call those bugs "shallow"), but many bugs are not easy to anticipate, or not easily triggered, or not easy to detect unless you know just what to look for (call those bugs "deep"). Testers are not just looking for shallow bugs, but for all *important* bugs. To find deep bugs requires insight of a kind that often comes only after playing with the product for an extended period of time. It may require noticing something strange and following-up on it. It may require doing complex operations that are easy for a human to do, yet expensive to automate. No good tester has all the good testing ideas right at the beginning of a project. Meanwhile, to automate something requires that you have a specific formal procedure that will spot every kind of important bug. There is no such thing. Big bugs don't follow any pre-ordained set of rules, and without rules we don't know what code to write to find them.

One more reason we can't just automate all the testing is that testing requires social competence. A tester must make judgments about what kind of data matters, what kinds of problems matter, which

testing is more important or less important, etc. These decisions are based on an understanding of the shifting and evolving values of the stakeholders of the product. These decisions must be defensible. All that requires social competence.

Automated checking only succeeds as a substitute for testing where there are no deep bugs that matter. Wherever you find that, it's usually in a very stable codebase or in a product that has customers who are tolerant about failure.

Suggested response: "Look. We can't automate parenting, management, getting to know people, falling in love, grief counseling, medical care, government, and no one is wondering when we can fire all the programmers and replace them with automated programmers. So, why do you think a skilled activity like hunting for a massive and completely unexpected bug is an exception to that? We can automate lots of interesting things that are part of testing, however. Wherever we can do that in a cost-effective way, we should."

"Why does testing take so long? How much testing is enough?"

These questions can only be answered in context. Some testing takes longer, and some testing can be done quite quickly. It depends on many factors, all of which can be called aspects of testability. See the *Heuristics of Testability* document for details.

Suggested response: "Which specific testing are you talking about? If we are talking about something specific, then we can reach a specific answer. Otherwise here is the general answer: testing takes however long it takes for the tester to build a compelling enough case that the product is tested well enough so that the stakeholders are able to make decisions based on a good enough knowledge of the risks associated with the product."

"Why do we need dedicated testers? Why don't we just get user feedback instead of testing? Why can't developers do the testing? Why don't we just get everyone to test?"

Often the foundational issue that gives rise to this question is a lack of understanding about what skilled testing is and what skilled testers do. The questioner may consider testing skills to be ubiquitous. The questioner may really be saying "since anyone can test as easily as anyone else, why bother with full-time testers?"

However, it may be that the lack of understanding is about roles: maybe the questioner is questioning the very idea of having people specialize or focus on any one activity, rather than fluidly moving from one activity to another over time. See the *How to Think About Roles and Actors* document for a list of dynamics that affect roles and the people who play them. For instance, one benefit of a having a person specialize in a role is *readiness*. If you only test once-in-a-while, you are probably not looking ahead and planning and preparing to perform testing. While a dedicated tester would be preparing, a casual tester if doing some entirely different job.

Suggested responses: "Dedicating people to a complex task improves their efficiency. This involves the improvement of competence, commitment, readiness, and coordination. Of course, in a low risk situation, we might not need such efficiency and effectiveness. Yes, we can get feedback from users, too, and we should, but you still need people who can process that information. Users are terrible bug reporters, and developers don't have the time and usually not the patience to follow-up on all those

half-baked reports. Certainly, everyone on the team can help testing. But someone needs to be responsible for it, or it will turn into chaos.”

Investigating Bugs: A Testing Skills Study

By James Bach
Principal Consultant, Satisfice, Inc.

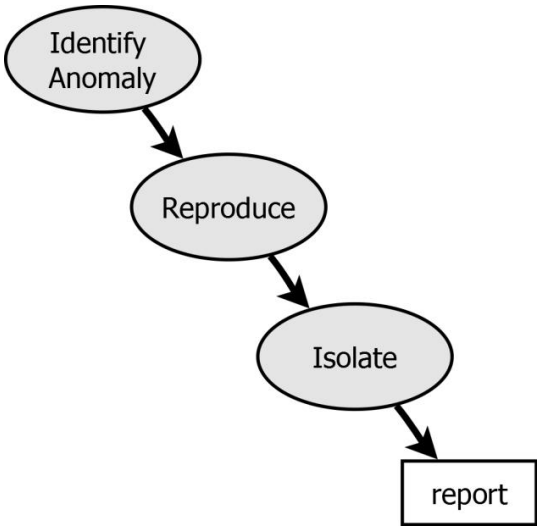
Ask any experienced tester how he does his work, and the answer is likely to be extremely vague ("Um, you know. I use my experience to... Um... black box the test case plan and such..."), or extremely false ("Our testing consists of detailed formal test procedures that are derived from written requirements"). Forget about bad testers, even *good* testers are notoriously bad at explaining what they do. Doing testing, describing testing, and teaching testing are all different things. No wonder that the IEEE testing standards are a joke (a very old joke, at this point), and based on talking with people involved in the upcoming ISO standard, it will be no improvement.

If we truly wish to develop our craft toward greater professional competence and integrity, then before we can worry how testing should be, we must be able to say how it is. We must study testers at work. Let me illustrate.

Years ago, I was hired by a company that makes printers to help them develop a professional testing culture. Instead of bringing in all my favorite testing practices, I started by observing the behavior of the most respected testers in their organization. I divided my study plan into segments, the first of which was bug investigation.

The organization identified one team of three testers (two testers and a test lead) that had a great reputation for bug investigation. This team was responsible for testing paper handling features of the printer. I wanted to see what made them so good. To get the most accurate picture of their practices, I became a participant-observer in that team for one week and worked with them on their bug reports.

Stated Procedure for Investigating Bugs



I sat down with George, the test lead, and asked him how he did bug investigation. He said, "Don't ask me to change anything."

"Good news, George," I replied. "I'm not that kind of consultant." But after some feather smoothing, he did answer:

1) *Notice if automation checks flags a problem.* The automation system alerts the testers that something did not perform as expected.

2) *Reproduce the problem.* The tester executes the test again to recreate the symptoms of the problem.

3) *Isolate the problem.* The tester edits the test script, cutting it down to the minimum required to exhibit the problem.

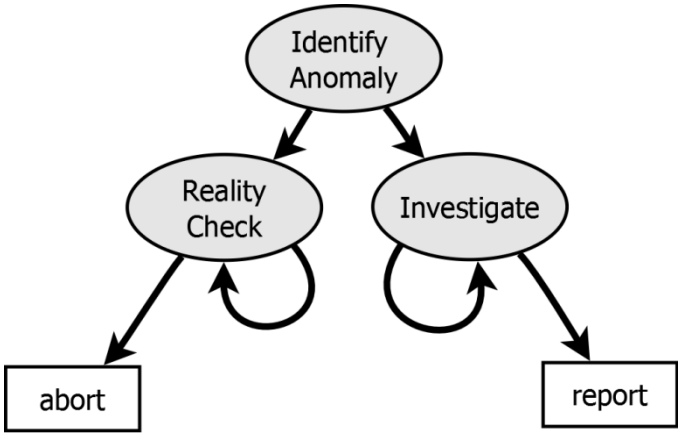
4) *Pass the problem to the test lead for investigation and reporting.* The tester delivers the edited test script to the test lead. The test lead investigates the problem to determine, as best he can, its dynamics and causes. The test lead then writes the bug report and submits it.

This description is typical of how testers claim to investigate and report bugs. It's only unusual in that the test lead performs the investigation and writes the actual bug report. However, there was another way in which this description was unfortunately typical: *it's not true.*

I knew it couldn't be true, because it's a process description that anyone in that company would claim to use, yet only this team was respected for the quality of its work. There must be something more to their process that I wasn't being told.

Sure enough, during my observations and further conversations with the team, I found the actual process used in the team to be much more sophisticated and collaborative than their stated process. Their actual process ranks among the best I have seen at any company. (I first said that in the year 2000, and it remains true in the year 2011).

Observed Process of Investigating Bugs



What I observed in practice was an exploratory investigation carried out by the whole team. When an anomaly worth investigating was spotted by a particular tester, the other two came over and they engaged together in two parallel loops of inquiry: *investigating the bug* and *questioning the status and merit of the investigation itself*. There were two possible outcomes, aborting the investigation or reporting the bug. The actual bug report was written by the test lead.

Part I: Identification

1. Notice a problem (during an automated check or any other situation).

   It is a general practice in the industry to construct tests that have specific expected results. This team took that idea further. Although they did establish certain specific expectations in advance, they also looked at whatever happened, as a whole, and asked themselves if it made sense. This extended even to the pattern of clicks and whirrs the printer made as it processed paper, and the timing of messages on the control panel. I call this the *Explainability Heuristic*: any unexplained behavior may be a potential bug, therefore we attempt to explain whatever happens.

2. Recall what you were doing just prior to the occurrence of the problem.
3. Examine symptoms of the problem without disturbing system state.

   Prior to starting a full investigation of a problem that may be difficult to reproduce, the testers capture as much volatile information as they can about it. This includes reviewing their actions that may have triggered the problem and examining the problem symptoms while disturbing the state of the printer as little as possible.

   During identification, the testers transition from a defocused behavior of observing whatever might be important to the focused task of investigating specific states and behaviors.

Part IIa: Reality Check Loop

The tester must decide whether to pursue the investigation or move back into open testing. So, prior to launching a bug investigation, and repeatedly during the investigation, these questions are asked.

1. Could this be tester error?
2. Could this be correct behavior?
3. Is this problem, or some variant of it, already known?
4. Is there someone else who can help us?
5. Is this worth investigating right now?
6. Do we know enough right now to report it?

The test lead is usually consulted on these questions before the investigation begins. But testers apply their own judgment if the test lead is not available. Contrary to George's first description of how his team worked, the testers on his team used initiative and routinely made their own decisions about what to do next.

In the event that an investigation is suspended because of the difficulty of reproducing it, it may still be reported as an intermittent problem. Whether or not it's reported, the testers will preserve their notes and be on the lookout for the problem as they continue testing. Some investigations go on like that for months.

Part IIb: Investigation Loop

Once it's determined that the anomaly is worth looking into, the investigation begins in earnest. As I observed them, bug investigations were not a linear execution of predefined steps. Instead, they proceeded as an exploratory process of gathering data, explaining data, and confirming explanations. The exploration was focused on reproducing the problem and answering certain key questions about it. When they were answered well enough, or when the amount of time and energy spent on the problem exceeded its importance (that's what the reality check loop is all about) the investigation ended and the bug was reported. Sometimes investigations continued after making an initial report. This was done so that the developers could begin to work on the bug in parallel with testers' efforts to give them better information.

The investigation process is marked by a series of focusing questions that are repeatedly asked and progressively answered:

1. How can the problem be reproduced?

The testers not only reproduce the problem, they try to find if there are other ways to make it happen. They progressively isolate the problem by discerning and eliminating steps that are not required to trigger it. They look for the most straightforward and general method of making it happen. They also seek to eliminate special conditions or tools that are not generally known or available, so that anyone who reads the bug report, at any later time, will have the ability to reproduce the problem.

2. What are the symptoms of the problem?

Apart from identifying and clarifying its obvious symptoms, the testers are alert for symptoms that may not be immediately obvious. They also look for other problems that may be triggered or exacerbated by this problem.

3. How severe could the problem be?

The testers try to analyze the severity of the problem in terms of how it would affect a user in the field or create a support issue for the company. They look for instances of the problem that may be more severe than the one originally discovered. They look for ways to reproduce the problem that are most plausible to occur in the field.

The testers also consider what this kind of problem may indicate about other the potential problems not yet discovered. This helps them assure that their test process is oriented toward areas of greatest technical risk.

4. What might be causing the problem?

The most interesting element I observed in the team's process of bug investigation is their application of technical insight about printer mechanisms (both hardware and firmware) to guide their investigation of the problems. In the course of investigation, the testers did not merely manipulate variables and factors arbitrarily. They investigated systematically based on their understanding the most likely variables involved. They also consulted with developers to refine their understanding of printer firmware dynamics.

Although there is no set formula for investigating problems, I observed that the testers relied upon their knowledge of printer mechanisms and their experience of past problems to organize their investigation strategy. So, maybe that's the formula: learn about how the printer works and pay attention to patterns of failure over time.

Part III: Reporting

Although I participated in bug investigation, I did not personally observe the process of writing a bug report in this team. The testers reported that they sometimes wrote a draft of the bug report themselves, but that all reports were edited and completed by the test lead.

Supporting Factors that Make the Process Work

Bug Investigation Philosophy

Apart from the process they follow, I found that there was a tacit philosophy of bug investigation in the team that seems to permeate and support their work. Here are some of the principles of that philosophy:

- We expect testers to learn the purposes and operational details of each test.
- We expect testers, over time, to gain a comprehensive expectation of the behavior of the product, and to follow-up on any anomalous behavior they detect at any time.
- Each bug is investigated by all members of the team.
- Bug investigation is primarily our job, not the developers. If we do our job well, then developers will be able to do their jobs better, and they will respect us for helping them.
- Testers should develop and use resources and tools that help in bug investigation.
- Ask for help. Someone else may know the answer or have an important clue. Seek advice from outside the team.
- We expect testers to use initiative in investigation and consult with the test lead as they go.

Individual Initiative and Team Collaboration

During the period I observed, the testers in the team treated each bug investigation as a group process. I had seen this before, and rarely since. They also consulted with testers outside their team, and with developers. Their attitude seemed to be that someone in the next cube may have information that will save them a lot of time and trouble.

The testers also showed personal initiative. They did not seem worried about crossing some forbidden line or running afoul of some corporate rule during the course their investigation. They appeared to take ownership of the problems they were investigating. The test lead told me that he

encouraged initiative in his testers, and that he expected the testers, over time, to learn how all the tests worked and how the printers worked. In separate interviews, the testers confirmed that sentiment, and stated that they felt that the resulting working conditions in their team were better than in most other teams they had served on at that company.

Observed Skills

I saw each of the following skills exhibited to some extent in each of the testers in the team. And in my opinion, the method of the investigation used in the team requires competence in these skills.

- *Skepticism.* Skepticism might be called the fear of certainty. It can be seen as central to the challenge of thinking scientifically; thinking like a tester. Good testers avoid sweeping claims about the product, because any claim may be refuted with the execution of the next test.
- *Performing an open investigation.* An open investigation is a self-managed investigation with general goals and few explicit constraints. An open investigation involves coordinating with clients, consulting with colleagues, collecting information, making conjectures, refuting or confirming conjectures, identifying issues, discerning and performing tasks, and reporting results. An open investigation in conjunction with testing is commonly called "exploratory testing."
- *Understanding external and internal product functionality.* Bug investigation requires a sufficient understanding of both external and internal workings of the technology. This knowledge is gained over time and over the course of many investigations, and through studying documentation, exploratory testing, or by observing other testers at work.
- *Consulting with developers or other testers.* Vital information needed to investigate problems is scattered among many minds. Good testers develop an understanding of the network of people who may be able to offer help, and know to approach them and efficiently elicit the information they need. In the case of developers, testers need the ability to discuss and question software architecture.
- *Test factoring.* When anomalous behavior is observed in the product, the ability to isolate the factors that may be causing that behavior is at the heart of the investigation process. This includes insight about what factors may be causally related, the ability to form hypotheses about them and to relate those hypotheses to observable behavior.
- *Experiment design.* Testers must be able to reason about factors and find methods of controlling, isolating, and observing those factors so as to corroborate or refute hypotheses about the product.
- *Noticing problems.* A tester can know how the product should function and yet still not notice a malfunction. Being alert for problems, even in the middle of investigating other problems, and even in the absence of an explicit and complete specification, is a skill by itself. This requires a good knowledge of applicable oracles, including tool-based oracles.
- *Assessing problem severity.* This requires understanding the relationship between the technology, the hypothetical user, the project situation, other known problems, and the risk associated with problems that may lie hidden behind the one being investigated. This skill also requires the ability to imagine and articulate problem severity in terms of plausible scenarios.

- *Identifying and using technical documentation.* Bug investigation often requires spot learning about the product. With printing technology, that can mean poring through any of thousands of pages of technical documents. Testers need to know where and how to find relevant information.
- *Recording and maintaining information about problems.* The testers must deliver information about a problem in an organized and coherent form in order for the test lead to confirm it and write the report. This includes the ability to make and maintain notes.
- *Identifying and using tools.* Tools that may aid testing are scattered all about. Enterprising testers should be constantly on the lookout for tools that might aid in the execution of tests or diagnosis of problems. Testers must have the ability and initiative to teach themselves how to use such tools.
- *Identifying similar known problems.* In order to know if a similar problem is already known, the testers must know who to check with and how to search the bug tracking system. This also requires enough technical insight to determine when two apparently dissimilar symptoms are in fact related.
- *Managing simultaneous investigations.* Rarely do we have the luxury of working on one thing at a time. That goes double when it comes to investigating intermittent problems. Such investigations can go on for weeks, so testers must have the ability to maintain their notes and report status over the long term. They must be able to switch among investigations and not let them be forgotten.
- *Escalation.* Since these investigations are largely self-managed, it's important to know when and how to alert management to issues and decisions that rightly belong at a higher level of responsibility.

Case Study: The Frozen Control Panel

This is an example of an actual investigation in the team that took place while I watched. It appears to be typical of other investigations I had been told about or personally observed. The important aspect of this case is not the conclusion— we could not reproduce this problem— but rather the initiative, teamwork, and resourcefulness of the testers. This investigation is documented in as much detail as we could remember in order to provide a feel for richness and flow of an exploratory testing process.

1. While Clay was running one of the paper handling tests, he encountered a printer lockup. Clay called Ken and James over to observe and assist.
2. Clay had been running an automated check that included many steps. After executing it once he started it again. This time, it began executing, then stopped, apparently waiting for a response from the printer. At that point Clay noticed that the printer was frozen.
3. Clay asked Ken if he knew about the problem and whether he thought the problem was worth investigating.
4. Without resetting the printer, Ken examined the surrounding symptoms of the problem:
    - Check control panel display (display showed "Tray 5 Empty" continuously).
    - Check ready and data light status (both were lit and steady).
    - Open and close a tray (display did not react; engine lifted the tray).
    - Open and close a door (display did not react; engine performed paper path check).

- Try control panel buttons (display did not react to any buttons).

5. Ken and Clay examined the test output in the terminal window and discovered that the test harness tool had stopped during its initialization, before any script code had been executed.
6. After a brief conference, Ken and Clay decided that the problem was worth investigating and conjectured that it may be due to an interaction between the timing of control panel display messages and messages sent to the printer.
7. Ken performed a cold reset of the printer.
8. Clay restarted the test tool. The problem did not recur.
9. Clay edited the test script down to the last few operations. He executed the modified script several times. The problem did not recur.
10. To test the hypothesis that the problem was related to the timing of alternating "READY" and "TRAY 5 EMPTY" displays on the control panel, Ken and Clay coordinated with each other to start executing the test tool at various different timings with respect to the state of the control panel display. No problem occurred.
11. We went to see a firmware developer on the control panel team, and asked him what might account for these symptoms. He seemed eager to help. He suggested that the problem might be a deadlock condition with the engine, or it might be a hang of the control panel code itself. He also suggested that we review recent changes to the firmware codebase, and that we attempt to reproduce the problem without using the test tool. During the course of this conversation, the developer drew some basic architectural diagrams to help explain what could be going on. We questioned him about the dynamics of his diagram.
12. The developer also conjectured that the problem could have been leftover data from a previous test.
13. Then we went to see a fellow who once supported the test tool. With his help, we scrolled through the source code enough to determine that all the messages displayed by the tool before it halted were issued prior to contacting the printer. Thus, it was possible that whatever happened could have been triggered by the first communication with the printer during tool initialization. However, we were unable to locate the tool routine that actually communicated with the printer.
14. Because the control panel locked up with the data light on, we knew that it was unlikely to have been in that state at the end of the previous successful test case, since that case had reported success, and left no data in the printer.
15. We looked for a way to eavesdrop on the exact communication between test tool and the printer, but found there was no easy way to do that.
16. We called upon another tester, Steve, for help, and together we wrote a shell script, then a Perl script, that endlessly looped while executing the test tool with an empty script file. At first we thought of introducing a random delay, but Clay argued that a fixed delay might better cover the timing relationships with the printer, due to the slight difference between the fixed time of the test and the presumably fixed response time of the printer.
17. We ran the script for about an hour. The printer never locked up.
18. While watching the control panel react to our script, Clay saw a brief flicker of an unexpected message on the display. We spent some time looking for a recurrence of that event, but did not see one.

19. We then went to visit a control panel tester to get his ideas on whether a problem like ours had been seen before, and how important a problem it could be. He advised us that such a problem would be quite important, but that he knew of no such problem currently outstanding.
20. Clay independently searched the bug tracking system for control panel problems, and found nothing similar, either.
21. After several hours of all this, we were out of easy ideas. So, we called off our investigation until the test lead returned to advise us.

The Study of Skill is Difficult

It's quite difficult to study the anatomy of a practice, and the skills that practice requires. You can't know at first exactly what to watch, and what to ignore. Anthropologists learn to watch behavior for long periods of time, and to relentlessly consider the possibility of researcher bias. And just the act of studying a set of skills makes people nervous and possibly change their behavior. I had to agree not to release any information about the progress of my study until I cleared it with the people I was studying.

Still, even a modest one-week study like this one can have profound positive effects on the team. When I gave this report to the team for approval, the team was a bit stunned at how much my description differed from their self-description. One of the testers asked me if he could staple it to his résumé. Perhaps there is even more depth to the skills of bug investigation than I have identified so far, but this is the sort of thing we must begin to do in our field. Observe testers at work and go beneath the grossly general descriptions. See what testers really do. Then maybe we can truly begin to build a deep and nuanced vision of professional software testing.

Rapid Software Testing Guide to Making Good Bug Reports

By James Bach and Michael Bolton, v.1.5

Bug reporting is central to testing. The bug report, whether oral or written, is the single most visible outcome of testing. And the quality of your reports is probably the single most important factor that determines your credibility as a tester. This document describes bug reporting from the perspective of Rapid Software Testing methodology.

Throughout this guide I will distinguish between what is expected of "**supporting testers**" (anyone helping the test effort temporarily or intermittently but not committed to the testing role or perhaps even that project) and "**responsible testers**" (those who assume the role of tester on a project and commit themselves to doing that job well over time).

What is a bug?

A *bug* is *anything about the product that threatens its value* (in the mind of someone whose opinion matters). Less formally, you could say that a bug is something that bugs someone whom we care about. Sometimes these are called *defects*, but I don't recommend using that term, since it sounds accusatory, and it implies that testers may only report things that they are absolutely sure are wrong with the product. Bug is a time-honored and somewhat vague term, which suits our need to report things that are possible threats to the value of the product, even things we might be wrong about.

Bugs are not about our own personal feelings. You can use your feelings to find a bug, but then you better know a good reason why someone else would have the same feelings. That's because your feelings don't really matter. A tester is an agent for the people who *do* matter, usually, because the people who matter probably have other things to do than test software. Thus, we must learn about the people who do matter: the users, the customers, the bosses, society, and the standards that are relevant to *them*. If you want to write successful bug reports, get inside the minds of your clients.

Normal Bug or Enhancement Request?

Bugs come in two flavors: normal or enhancement. A normal problem is a failure of the product to fulfill its intent; whereas an enhancement request is for when you believe the intent itself should be changed. In other words, "the product isn't doing what you want it to do" is a regular bug; "the product is doing what you want, but you should want something better than that" is an enhancement request. These different cases must be reported a little differently. In the case of an enhancement request, you are making a pitch to the stakeholders to aim higher (in your opinion). You are essentially playing designer, here, which means if you aren't careful you will step on the toes of the designer.

Whenever you make an enhancement request, I suggest phrasing the bug title as a request (i.e. start with the word "please…") to make it clear that you are suggesting a change to the scope of the product.

What is a bug report? What are its elements?

A *bug report* is a description of a suspected bug. The most basic bug report is a statement to the effect that "here's something I think may be wrong with the product." In real life this could manifest in a manner as simple as pointing at a screen and saying "Uh oh, look at that." In fact, that may be all you need to do in the case where you are testing for a friend standing next to you and you both have a strong shared mental model of what the product should be and do. If we are all close friends or if we all belong to the same hive mind, bug reporting can be pretty easy.

Bug reports can be formal or informal, written or oral. Underlying even the simplest bug report is a certain structure with the following four elements:

- **A description of the problem you perceive.** What bad thing happens; or what good thing doesn't happen? Be specific and clear about that. Ask yourself if that is the root of the matter, or whether there is something bigger or more fundamental that you ought to report instead.
- **How you encountered that problem.** The bug you perceive ought to be grounded in a direct observation of the product itself. Be specific about steps and data you used.
- **The reason why it is a problem.** The means by which you recognize a problem that you encounter while testing is called an *oracle*[1]. It can be a principle, specification, feeling, example, tool, or even a person. All bug reports are based on some sort of oracle, and maybe several different oracles. Some oracles are more authoritative (stronger), others merely suggestive (weaker). You might use a weak oracle (such as a feeling that "this is hard to use") that gives you a suspicion that something is not right and needs investigation, then after some investigating find that you have a stronger oracle ("this violates the defined usability standard) against which to actually report the bug.
- **Why it's a problem that *matters*.** Just the fact that a behavior is a threat to the value of the product is not necessarily interesting. Your clients need to know: is it a big bug or a little bug? You should be ready to say how important a bug it could be. This is related to how likely it is to be seen and how much damage it could do if it occurs. More on that below.

Even if you make a very informal bug report, be ready to make a more complete and explicit report, if your report is challenged. Some of the common ways bug reports are challenged include:

- "I don't know what you are talking about."
- "That doesn't happen when I try it."
- "I don't see why that's a problem."
- "That's only a problem for beginners."
- "It's a problem but it's difficult to fix, and there's an easy workaround."
- "It's a problem but only weird users and testers will ever stumble into it."
- "Maybe you don't like the way it works, but most real users will like it."

You ought to take challenges like these in your stride. Remember that the developer has a fundamentally optimistic, builder's mentality. This is a good thing. To create anything complex and wonderful requires optimism. Your bug reports are like ants raining on their picnic. So, keep your cool, and be ready to offer evidence or argument to support the best case you can make that a bug is worth fixing.

Apart from challenges, you also should anticipate the common questions that developers and managers may ask, such as:

---

[1] See Michael Bolton's five-part series on oracles: http://www.developsense.com/blog/2015/09/oracles-from-the-inside-out/

Basic Information (expected from anyone)

- What seems to be the problem?
- What exactly did you see happen?
- What were you doing right before you saw the problem? Were you doing anything else interesting at the same time?
- Have you seen the problem more than once? If so, what other times have you seen it?
- Did you take any screenshots or a video?
- What data were you using? What files? What exactly did you input?
- What good reason do you have for thinking it is wrong?
- What version of the product were you testing? What platform are you using?

Investigation Details (expected from responsible testers)

- Is it already reported?
- Have you tried to reproduce the effect? What are the steps to reproduce it?
- Have you tried simple variations of the input or operations that led to the symptoms? Have you tried on different platforms or in different modes?
- Have you checked to see that the product is actually installed properly and running? Is your system is properly configured?
- Did this problem exist in the earlier versions of the product?
- Do you have a specific suggestion of what should happen instead? How could it be better?
- How easy is it to make the bug happen? What is the worst consequence you can think of that could result from it?

Not all of these questions apply for every bug, but your credibility as a tester depends on being able to answer them in the cases where they do apply. For instance, if you report that "uploading a big file" causes some sort of error, then you must say in the bug report exactly how big were the files that you tried, and be sure those exact files are accessible to the developers.

Bug Investigation

Supporting testers are generally not expected to investigate bugs beyond what is necessary to make a clear report.

Responsible testers are expected to investigate bugs to the point they can make clear, relevant, and concise reports. However, for puzzling or difficult to reproduce bugs, it is often is the case that the developer will have immediate insight into the underlying causes. For that reason, I recommend the *10-minute heuristic*: investigate a puzzling bug for no more than about 10-minutes before checking in with a developer about it (assuming the developer is available). If the developer is also puzzled, continue the investigation, otherwise, if the developer claims to know what is happening, just report what you know and move on to find the next bug. That will mean you are finding more bugs instead of making ever nicer-looking reports.

The goal of bug investigation is to gather good enough information about it so that your clients can evaluate the problem and fix it. This generally means four things:

- **Reproduce** the problem reliably.
- **Isolate** the bug by identifying and eliminating extraneous factors, discovering the limits of the bug, and collecting evidence about its underlying causes.
- **Generalize** the report by discovering the broadest occurrence and impacts of the bug.
- **Support** your case with relevant and necessary data that will make the developer's fix investigation easier.

Bug investigation often requires technical knowledge, product knowledge, and analytical skills that are beyond the scope of this guide to explain. That's why we don't expect much investigation to be done by **supporting testers** coming in to the project temporarily.

Formal vs. Informal Bug Reporting

Consider three kinds of bug reporting: MIP'ing, formal reports, and black flagging:

- **MIP.** This stands for "mention in passing." To MIP a bug is to report it in a sentence or two by voice, email, or instant message. It can even take the form of a question ("is this supposed to work this way…?). There is no expectation of formally tracking such reports, and there is no template for them. The main reason to MIP a bug is when you are fairly confident that what you are reporting is *not* a bug, and yet you have a suspicion that it could be. MIP'ing is partly a strategy for learning about the product, since early on in testing many things that look like problems to you might not be. MIP'ing helps preserve your credibility because if a MIP'ed bug turns out to be a real issue, you can fairly say that you did report it, and because if it is not a bug, you can fairly say that you didn't create unwieldy paperwork for the team. MIP'ing is a very good way to work when you are pairing with a developer for early testing.

  MIP'ing is an excellent protocol for bug reporting when running a mass testing event with **supporting testers** when developers or other experts are in the room with them. The supporters then serve as "bug scouts", while the experts perform investigations and take responsibility for formal reporting, if required.

- **Formal Reports.** This means recording written bug reports in some sort of tracking system. Formal bug reporting is slower and more expensive than MIP'ing, but has obvious advantages when you are dealing with large numbers of bugs. Expecting **supporting testers** to do formal bug reporting may not be reasonable, but if they do, someone with responsibility will need to edit them. Poor quality formal bug reports can erode the credibility of the testing effort.

- **Black Flagging.** This means going beyond reporting a bug to the extent that testers raise an alarm about a serious underlying problem in the development effort. This may be necessary for safety or security bugs that can only occur when there is a breakdown of development discipline. Black flagging is for when you suspect that a bug is part of a much larger group of bugs that may not have been found yet, or may not yet have been put into the code.

These forms of bug reporting are not the only forms that exist. But they serve to illustrate that bug reporting is a socially-situated activity. However or whatever ever you do with your reporting, your process must ultimately fit the social milieu of the project.

Elements of a Basic Formal Bug Report

Here are the most common fields found in a formal bug report:

- **Title.** A short summary that expresses the essence of the bug.
  - One sentence long; about twelve words or less.
  - It must be distinctive, not generic (i.e. don't write "the product crashes"). The title should uniquely identify the bug among all the other bug report titles.

  - Try to put the actual problem right at the beginning (e.g. "Crash when logging in as admin" rather than "When logging in as admin, product crashes") because it's easier to read when looking at a list of bugs. As journalists say, "don't bury the lead."[2]
  - If it is an enhancement request, consider starting the title with the word "please."
- **Description.** Any other information about the specific failure mode and behavior of the system that members of the team need to know about the bug.
  - Keep it short. Give reasonable details about the bug, but don't include information that everyone on the team will certainly already know. If the problem is very obvious (e.g. "Company name spelled wrong on the home page") then you hardly need to write a description.
  - Write in professional language. Don't use texting jargon. Spell words correctly.
  - Provide steps to reproduce *if those steps are not obvious*. Don't provide steps that are obvious (e.g. "1. Connect to the Internet 2. Start the browser").
  - Indicate your oracles. That means say why you think this is a bug, *unless this is obvious*. Don't say silly things like "the product should not crash." That sounds insulting and it adds nothing to the report.
  - Note any workaround for the bug that you know about.

- **Version.** This is the latest version of the product in which you detected the bug. If you also tested earlier versions, note that in the description.

- **Environment.** The platform you were testing on. Typically this is your hardware, browser and operating system. If you are testing an online product, specify the server. Report any environment element that is interesting or unusual, or if it is customary to report it.

- **Attachments.** For all but the easiest to understand bug reports, it will be much appreciated if you attach screenshots of the problem,[3] or even small videos. Also include links to any critical data files needed to reproduce the problem.

In addition to the basic fields, your bug tracking system may have other fields, as well. It will autofill the ID, Reporter, and Date Reported fields. Then there is Status, Severity, and Priority, which follow a protocol that is specific to your company and your project, so I won't discuss them here.

Give the bug report a good focus.

- **Report the most compelling version of the bug.** Bug reporting is a kind of sales job. You must frame the report in the most compelling (yet truthful) way in order to maximize the chance that it will be understood and acted upon. Try to focus on the aspect of the bug that can have the most user impact, or the most negative public impact for your company. In other words, try to identify the strongest, most compelling oracle that you can.
- **Avoid reporting multiple problems in one bug report.** Unless multiple problems are probably the symptoms of one underlying fault in the product, they should be separated into distinct bug reports. This is because it's very easy for a developer to fix one problem, while accidentally forgetting to fix others that are listed in the same report.

---

[2] https://www.merriam-webster.com/words-at-play/bury-the-lede-versus-lead

[3] Andrea Hüttner, a tester from Germany who works with a multilingual team, points out that videos and screenshots are particularly important for her team because they communicate well across language barriers; and even if everyone "speaks the same language," videos and picture can bridge gaps in vocabulary.

- **Avoid reporting the same problem in multiple reports.** It is often difficult to tell whether two problems that seem similar are genuinely the same problem. So, make your best guess or consult with the developer to be sure.

Assessing the Significance of a Bug

A tester is the first judge of how "big" the bug is. This is true even for **supporting testers**, to some degree. But for **responsible testers** it is a very important part of your work.

What makes a bug important? Basically four things:

- **How frequently does it appear; or how easy is it to make happen?** A bug that is seen often or by a lot of users is going to be more important, all other things being equal. Are there lots of different kinds of events that can trigger the bug? Is it highly vulnerable to the triggering events? How visible and obvious is it when it appears?
- **How much damage does it do when it occurs?** The most important bugs are generally the ones that stop the project, itself: so-called *blocking bugs*. These are bugs that prevent you from testing. Down from that are bugs that harm or block the user. A bug that deletes data may be more important than one that merely creates confusion in the user interface, but the opposite can also be the case when confusion could result in dangerous user behavior. While there are no hard rules about what specific symptoms constitute "more damage," try visualizing the problem, then consider the importance of the user who is affected, and how upset they may be because of the bug.
- **What does the bug imply about other potential risks?** A bug may be especially important because it implies that there is a big problem in the development process itself that may have resulted in many similar bugs that are not yet found (see "black flagging", above).
- **What bad publicity might come from the bug?** Bad feelings and bad reputation can accrue from bugs even if the objective severity is not that bad. Consider how a bug might sound when people complain about it on social media. Consider how it might erode trust in your company's brand.

Common Mistakes Testers Make in Bug Reporting

Watch out for these problems in your reports:

- **Poorly worded title.** The title is rambling, incoherent, generic, too long, or otherwise not representative of the substance of the bug report.
- **Reporting an unsupported opinion.** A personal opinion with insufficient grounding or evidence to support it is no basis to report a bug. The tester is not an authority; the tester is an agent for people who are authorities.
- **Reporting something that is not a problem.** Even if the bug is based on an oracle other than personal opinion, it may still be an incorrect oracle. The product may actually be intended to work the way that it does.
- **Not enough information to reproduce.** There is not enough information to enable the developer to verify the existence of the problem.
- **User error.** The report is based on a mistaken test, such as when an observation or operation was not done properly.
- **Focusing the report on the wrong thing.** Sometimes testers will report a small problem that sits in the shadow of a much more important problem. That can happen when they don't take a moment to consider the bigger picture of the bug.

- **Disrespectful reporting.** The report is written in a manner that will irritate the developer or manager and erode credibility of the tester. This happens usually when the report is written in a sloppy way, or seems to denigrate the developer.
- **Pedantic reporting.** The report includes information that is common knowledge, implying that the developer is ignorant of basic facts. Example "Actual: product crashes. Expected: product doesn't crash." The second part of that is pedantic.
- **Terse report.** If there are not enough words, it's too hard to figure out what is being reported.
- **Unnecessary text.** Including information that is already common knowledge can make it seem like you are writing for the sake of filling in fields, rather than with the intent to communicate clearly.
- **Multiple reports.** More than one bug report packed into one record. When in doubt, break it out.
- **Getting the severity wrong.** Not all bug reporting systems require the tester to assess severity, but if you get it wrong you will either cause someone to do unneeded work or else let an important bug get buried.
- **Confusing an ordinary bug with an enhancement request.** Too often, I see testers reporting minor bugs as enhancement requests. This seems to occur more often when a tester has a specific fix in mind. It's okay to make a suggested fix, but that's not an enhancement request. Enhancement means that you are suggesting a change in product scope; a change in the requirements. Otherwise, you are merely offering a suggesting for how to correct an ordinary failure of the product to do what it is intended to do. Reporting a normal bug as an enhancement tends to decrease the probability that it will get fixed.

A Context-Driven Approach to Automation in Testing

By James Bach and Michael Bolton February, 2016 (v1.05)

There are many wonderful ways tools can be used to help software testing. Yet, all across industry, tools are poorly applied, which adds terrible waste, confusion, and pain to what is already a hard problem. Why is this so? What can be done? We think the basic problem is a shallow, narrow, and ritualistic approach to tool use. This is encouraged by the pandemic, rarely examined, and absolutely false belief that testing is a mechanical, repetitive process. Good testing, like programming, is instead a challenging intellectual process. Tool use in testing must therefore be mediated by people who understand the complexities of tools and of tests. This is as true for testing as for development, or indeed as it is for any skilled occupation from carpentry to medicine.

Thank you to our reviewers: Tim Western, Alan Page, Keith Klain, Ben Simo, Paul Holland, Alan Richardson, Christin Wiedemann, Noah Sussman, and Joseph Quaratella.

James Bach, james@satisfice.com, http://www.satisfice.com, @jamesmarcusbach
Michael Bolton, michael@developsense.com, http://www.developsense.com, @michaelbolton

A Context-Driven Approach to Automation in Testing

By James Bach and Michael Bolton February, 2016 (v1.05)

Table of Contents

In this white paper, we offer a vision of test automation that puts the tester at the center of testing, while promoting a way of thinking that celebrates the many things tools can do for us. We embrace tools without abdicating our responsibility as technical people to run the show.

Tools can be powerful, and we are going to say encouraging and helpful things about them. But automation can also be treacherous—not least because the label "automation" refers to a mess of different things. So, we must begin with a sober look at some basic misconceptions that add terrible waste, confusion, and pain to what is already difficult even in the best of times. If you need good testing, then good tool support will be part of the picture, and that means you must learn why we go wrong with tools.

Robots! Help!

We can summarize the dominant view of test automation as "*automate testing by automating the user."* We are not claiming that people literally say this, merely that they try to do it. We see at least three big problems here that trivialize testing:

1. The word "automation" is misleading. We cannot automate *users*. We automate *some* actions they perform, but users do so much more than that.
2. Output checking can be automated, but *testers* do so much more than that.
3. Automated output checking is interesting, but *tools* do so much more than that.

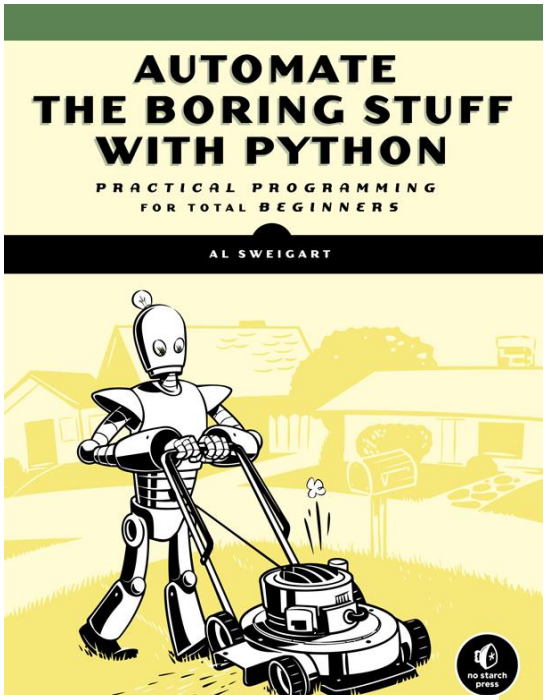


Figure 1

Automation comes with a tasty and digestible story: replace messy, complex humanity with reliable, fast, efficient robots! Consider Figure 1. It perfectly summarizes the impressive vision: "Automate the Boring Stuff." Okay. What does the picture show us?

It shows us a machine that is intended to function as a human. The robot is constructed as a humanoid. It is using a tool normally operated by humans, in exactly the way that humans would operate it, rather than through an interface more suited to robots. There is no depiction of the process of programming the robot or controlling it, or correcting it when it errs. There are no broken down robots in the background. The human role in this scene is not depicted. No human appears even in the background. The message is: robots replace humans in uninteresting tasks without changing the nature of the process, and without any trace of human presence, guidance, or purpose. Is that what automation is? Is that how it works? No!

Of course it is a light-hearted cartoon, not to be taken seriously. The problem is, in our travels all over the industry, we see clients thinking about real testing, real automation, and real people in just this cartoonish way. The trouble that comes from that is serious.

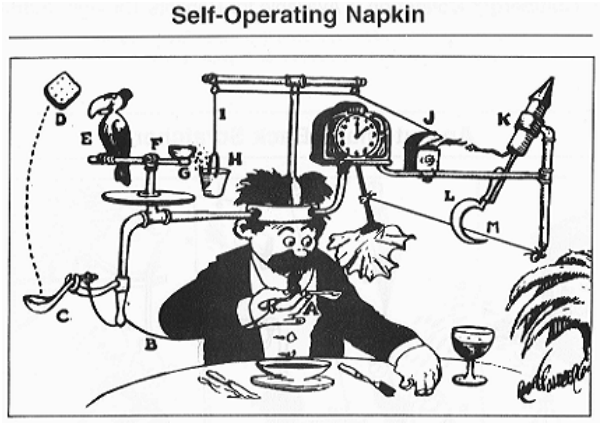


Figure 2: Cartoon by Rube Goldberg

How serious? In the experience of the authors, observing projects going back to the 80's, we find that it is normal for large scale automation efforts to lavish the bulk of their budgets in the detection of trivial and obvious GUI-level bugs, drawing much needed time and effort away from the hunt for serious but subtle problems—what we call *deep* bugs. Furthermore, the typical automation approach has the character of a Rube Goldberg machine—swimming in dependencies and almost comically prone to breakdown.[1] This sort of automation becomes almost like a new stakeholder on the project; as with some obsessive-compulsive "high maintenance" cleaning lady who won't even enter the house until it is already spotless. We believe the effort typically invested in automation would in most cases be better invested directly into humans interacting with the product in complex and sophisticated ways (which also finds the shallow bugs) and into less expensive supporting tools that help testers test better.

No one can deny that automation tool sales demos are impressive. What we deny is that people agree on what "automation" means, what it should be, and that those sales demos translate into practical value on ordinary projects.

The Trouble with "Automation"

The trouble with "test automation" starts with the words themselves. Testing is a part of the creative and critical work that happens in the design studio, but "automation" encourages people to think of mechanizable assembly-line work done on the factory floor.

The term "test automation" is also ambiguous. It is common to hear someone say a sentence like "run the test automation," which refers specifically to tools. A sentence like "test automation is worth doing" refers not only to tools but also to the enterprise of creating, maintaining, testing, and operating those tools. In the first sense, test automation is not human at all. It's incredibly fast and inexpensive, too, since you don't pay the computer. In the second sense, test automation is a skilled activity performed by humans who write and operate software over hours, days, or weeks—and those people must be paid for their time.

---

[1] In one case, James was called in to help a project that had "more than 3,000" automated scripts, developed over a nine-month period. James asked to see them executed, whereupon it was revealed that they *all* had been broken by a recent update to their expensive commercial test tool and ongoing updates to their own product.

We observe that in common parlance, the driving tactic of "test automation" is to script ordinary, rote actions of a user of the product—and a rather complacent, unimaginative user, at that— then get the machinery to punch the keys at dazzling speed, and then check to see whether specified actions and inputs produce specified outputs. From there, it's a small step to start thinking of "test automation" as a sort of tester in its own right. But even a minimally skilled human tester does far more than blindly adhere to prescribed actions, and observes far more than the output of some function. Humans have complicated lives, agendas, talents, ideas, and problems. Although certain user and tester actions can be *simulated*, users and testers themselves cannot be *replicated* in software. Failure to understand this simple truth will trivialize testing, and will allow many bugs to escape our notice.

"We define a tool as any human contrivance that aids in fulfilling a human purpose."

How can we think about all this more clearly?

First: Call them tools (not "test automation").

We define a tool as **any human contrivance that aids in fulfilling a human purpose**. A test tool could be software; hardware; a map, document, or artifact; or some other heuristic that aids in fulfilling a testing purpose. We are primarily concerned with software-based tools, here.

The term "test tool" connects us to the ordinary, everyday understanding that these contrivances do not work without human guidance; they extend the capabilities of an appropriately skilled human. Moreover, "tool" opens the door to the many ways that tools can lighten burdens and amplify the power of testers.

Meanwhile, the term "test automation" threatens to *dissociate* people from their work. To understand why, you must consider what testing is. To test is to seek the true status of a product, which in complex products is hidden from casual view. Testers do this to discover trouble. A tester, working with limited resources, must sniff out trouble before it's too late. This requires careful attention to subtle clues in the behavior of the product within a rapid and ongoing learning process. Testers engage in sensemaking, critical thinking, and experimentation, none of which can be done by mechanical means. Yet, in our long experience visiting hundreds of companies and teams, we find managers and technocrats who speak of testing routinely ignore these intellectual processes. We have tried reminding them—and our own colleagues, at times—of these crucial elements that cannot be encoded into test cases or test software. "Oh we agree," they might say, but then lapse back into speaking exactly as if the essence of testing is somehow expressed in their "test automation." After years of this struggle, we conclude that the term itself is a sort of narcotic.

Computer software is comprised strictly of explicitly encoded patterns. Any valuable or important pattern of behavior will not happen unless it is expressed in code. This is obvious. What is not so obvious is that much of what informs a human tester's behavior is tacit knowledge[2]. (Whereas, explicit knowledge is any knowledge that is represented as a string of bits, tacit knowledge is that which is not or cannot be so represented.)

When a human tester interacts with a product, he spontaneously reacts to an astonishing variety of surprising and erroneous events without ever having been consciously aware of an expectation about them. If, for instance, a window turns purple for a moment, or an extra line appears, or a process takes a little longer to complete one out of ten times, he almost effortlessly notices and reacts. But when this tester tells the story of this test, perhaps by writing down its steps and expected results, only a small part of all those real expectations are expressed. No tester will encode an *unconscious* expectation or *unanticipated* action. Since the testing humans actually do cannot be put into words, it cannot be encoded and therefore cannot be automated. We should not use a term that implies it can be.

"Since the testing humans actually do cannot be put into words, it cannot be encoded and therefore cannot be automated."

Everyone knows *programming* cannot be automated. Although many early programming languages were called "autocodes" and early compilers were called "autocoders," that way of speaking peaked around 1965[3]. The term "compiler" became far more popular. In other words, when software started coding, *they changed the name of that activity to compiling, assembling, or interpreting*. That way the programmer is someone who always sits on top of all the technology and no manager is saying "when can we automate all this programming?"

To produce high-quality products and services, we need *skilled people applying appropriate tools to fulfill the mission of testing.* The common terms "manual testers" or "automated testers" to distinguish testers are misleading, because all competent testers use tools. Programmers and researchers use tools, too, but no one speaks of "automated programming" or "automated research." No manager who calls for automated testing aspires to automate his management. The only reason people consider it interesting to automate testing is that they honestly believe testing requires no skill or judgment.

Since all testers use tools, we suggest a more interesting distinction is that *some* testers also *make* tools—writing code and creating utilities and instruments that aid in testing. We suggest calling such technical testers "toolsmiths." Although toolsmiths and their tools help to extend, accelerate,

---

[2] An excellent source for learning about this is *Tacit and Explicit Knowledge* by Harry Collins. We call Harry the "sociologist for testers" because his studies of scientists at work apply perfectly to the world of testing, too.
[3] According to Google Ngram Viewer

and intensify certain activities within testing, they do not automate testing itself. Therefore, from this point on, we shall try to avoid using the term "test automation."

Second: Think of testing as much more than output checking.

We say that testing is *evaluating a product by learning about it through exploration and experimentation, which includes to some degree: questioning, study, modeling, observation and inference, etc.* [4]

We choose our words carefully. Testing is necessarily a human process. Only humans can learn. Only humans can determine value. Value is a social judgment, and different people value things differently. Technologists may believe that they can automate the evaluation of requirements by encoding them into a script, but the evaluation is provisional and incomplete until it has been reviewed by a human. There are nearly always circumstances in which a manager will say "the tool is reporting a bug, but it is really not a problem in this case."

Exploration is central to our definition of testing because we don't know where the bugs are before we find them. Indeed, with any new product we must discover where to look for problems, and there are too many places to look for us to check them all. We don't even know for sure what counts as a bug; that is a judgment that drifts and shifts over the course of a project. We emphasize experimentation because good tests are literally experiments in the scientific sense of the word. At least 300 years before anyone ever wondered what software could or would do, "natural philosophers" were systematically testing nature via their experiments.[5] What scientists mean by experiment is precisely what we mean by test. Testing is necessarily a process of incremental, speculative, self-directed search.

Finally, the "etc." at the end is a signal that testing incorporates many other analysis-related activities and disciplines. Activities

Good Checking is a Subset of Testing

• • •

Checking is not the same as testing in the way that biting is not the same as eating; tires are not the same as cars; and spell checking is not the same as editing. Good checking is always a product of and embedded in a test process. Testing gives checking its value and meaning.

[4] See http://www.satisfice.com/blog/archives/856

[5] "…what these several degrees are I have not yet experimentally verified; but it is a notion, which if fully prosecuted as it ought to be, will mightily assist the astronomer to reduce all the Celestial motions to a certain rule, which I doubt will never be done true without it…" Robert Hooke, 1674, An Attempt to Prove the Motion of the Earth by Observations, (http://bit.ly/1MDwhBI)

that aren't themselves testing, such as studying a specification, become testing when done *for the purposes of testing*.

Let's break down testing further. What do we specifically do when we test? Testing is a performance that involves several kinds of ongoing, parallel activities:

- we *design* our testing by learning and modelling the product, determining test conditions to cover, generating specific test data, identifying and developing oracles (i.e. the means to recognize problems when we encounter them), and establishing procedures to explore and experiment.
- we *interact with the product* by configuring, operating and observing it.
- we *evaluate* the product by using appropriate oracles to detect inconsistencies between the product and qualities that we might consider ideal.
- we *record and report* the testing work that has been done.
- we *manage* the testing work, which includes understanding the current status of testing, analyzing product risk, scoping and assigning testing tasks.

All of these activities can be helped with tools.

Distinguish between checking and testing.

We find it necessary to distinguish between checking and testing. Checking is **the process of making evaluations by applying algorithmic decision rules to specific observations of a product**. This is different from the rest of testing in one vital way: *it can be completely automated*. Checking is an appropriate place to use that word "automation."

In testing, we design and perform experiments that help us develop our understanding of the status of the product. This understanding is an interpretation; an assessment. *But it is not a fact.* Simple facts are arguably "verifiable," but quality is never a simple fact. Quality is a working hypothesis. When you exercise software and fail to spot a specific problem, you have not proven or demonstrated that "it works." All you know is that you haven't yet recognized a failure. All you have demonstrated is that the product *can* work. The product may have failed in a subtle way you did not or cannot yet detect., Maybe it works fine now, but won't work ten minutes from now. So does it really, truly, deeply work? *No output check can tell you that. No collection of output checks can tell you that.*

Indeed, any advertiser, late-night TV pitchman, or stage magician can show you that something *appears* to work. Our job as testers is not to obey the ad, swallow the pitch, or believe the trick. Our job is to figure out what the ad leaves out, where the product doesn't meet the claims, or how the magician might be fooling us. Although routine output checking is part of our work, we continually re-focus on non-routine, novel observations. Our attitude must be one of seeking to find trouble, not verifying the absence of trouble—otherwise we will test in shallow ways and blind ourselves to the true nature of the product.

Evaluating quality is a task that requires skillful, complex, non-algorithmic investigation and judgment. That task can be supported and accelerated by tools, but it cannot be performed by the tools themselves.

Checking is important.

Good checking is a subset of testing. Checking is not the same as testing in the way that biting is not the same as eating; tires are not the same as cars; and spell checking is not the same as editing. Good checking is always a product of— and embedded in— the processes of designing, implementing, and interpreting those checks, which are *human* activities; which constitute testing. Testing gives checking its value and meaning. Whereas, checking keeps testing grounded.

Automated checking is a *tactic* of testing, and can have considerable value. Programmers who adopt automated checks into their coding practices can provide themselves with fast, inexpensive feedback. Checking through an API beneath the GUI level can be particularly useful. In designing such low-level checks, programmers and testers can profitably work together.

We are more doubtful of automated checking at the GUI level. GUIs are notoriously fussy. Because non-technical people can see them and discuss them, GUIs may change much more capriciously than the underlying interfaces that only programmers see. This can lead to a large, expensive maintenance effort just to keep the simple checks running. Moreover, GUIs are designed to feel natural and comfortable for people, not for other software. You may need a skilled full-time programmer to maintain all the code necessary to attempt to simulate a speedy but unskilled human tester. That is probably not a money-saving proposition.

Third: Explore the many ways to use tools!

The skill set and the mindset of the individual tester are central to the responsible use of tools. When we say this, however, some people seem to hear us saying that tools are not important, or that context-driven testers hate tools. *Nothing could be farther from the truth.*

Let's catalog some of the many ways tools help us in testing:

- **In design; we use tools to help us**
  - produce test data (tools like spreadsheets; state-model generators; Monte Carlo simulations; random number generators)
  - obfuscate or cleanse production data for privacy reasons (data shufflers; name replacers)
  - generate interesting combinations of parameters (all-pairs or combinatorial data generators)
  - generate flows through the product that cover specific conditions (state-model or flow-model path generators)

- **In product interaction, we use tools to help us**
  - set up and configure the product or test environments (like continuous deployment tools; virtualization tools; or system cloning tools)
  - submitting and timing transactions; perhaps for a long time; at high volume; under stress (profiling and benchmarking tools)
  - encode procedures like operating the product and comparing its outputs to calculated results (this is automated checking).
  - simulate software or hardware that has not been developed yet; or that we do not have immediately available to us (mocking or stubbing tools)
  - probe the internal state of the system and analyze traffic within it as testing is being performed (instrumentation; log analysis; file or process monitors; debugging tools)
- **In evaluation, we use tools to help us**
  - sort, filter, and parse output logs (text editors; spreadsheets; regular expressions)
  - visualize output for comparative analysis (diffing, charting and graphing tools, conditional output formatting)
  - develop, adapt and apply oracles that help us recognize potential problems (source file or output comparison tools; parallel or comparable algorithms; internal consistency checks within the application; statistical analysis tools)
- **In recording and reporting, we use tools to help us**
  - record our activities and document our procedures (note-taking tools; video-recording tools; built-in logging; word processing tools; user interaction recording tools)
  - prepare reports for our clients (mind maps; word processors; spreadsheets; presentation software)
- **In managing the testing work, we use tools to help us**
  - map out our strategies (mind maps, outline processors, word processors)
  - identify what has and has not been covered by testing (coverage tools; profilers; log file analysis)
  - preserve information about our products, and to aid other people in future support and development (wikis; knowledge bases; file servers)

And this is an *incomplete* list of the ways in which we use tools to help us. Moreover, we use tools to help us produce the tools that we use.

You have probably noticed how we repeatedly said “We use tools to help us...” We have chosen these words deliberately to emphasize once again that *tools don’t do testing work; tools help testers to do testing work*. In conversation about testing, tools may be important, but the center of testing must be the skill set and the mindset of the individual tester.

Let your context drive your tooling.

By "context", we mean **the set of factors that should affect the decisions of a responsible tester**. Generally speaking, a craftsman who has both the skills and intent to select and apply the appropriate tools and methods for any given context can be called *context-driven*. More specifically the Context-Driven school of software testing is a paradigm of testing based on the following principles:

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project's context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn't solved, the product doesn't work.
6. Good software testing is a challenging intellectual process.
7. Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.

These principles were written by Cem Kaner, James Bach, and Bret Pettichord, and first published in their book *Lessons Learned in Software Testing: A Context-Driven Approach*, which is the seminal book on Context-Driven thinking.

Note that if you are working in a way that solves the problems that exist in your environment, you may be doing context-*specific* work without necessarily being context-*driven*. To be context-driven you must be ready and able to change the way you work if and when the context changes. That's why the Context-Driven community focuses on developing skills and sharing experiences across many kinds of projects and technologies. This is why we foster peer conferences dedicated to conversation and debate.

How specifically does context drive tooling?

Context drives tooling through the activity of ongoing problem-solving. We do that by developing in our minds various understandings, including:

- what surrounds us and our place in that world
- our clients and our mission
- other people involved and what they are trying to do
- tools and techniques available to us
- actions we could take and the effects they may have
- the immediate costs (and time) of those actions
- the long term costs of those actions
- the value of learning from trying new things

These understandings may be thought of as spaces that we explore throughout our projects and careers. As we learn and grow, during the course of our projects and careers, we get better at navigating them.

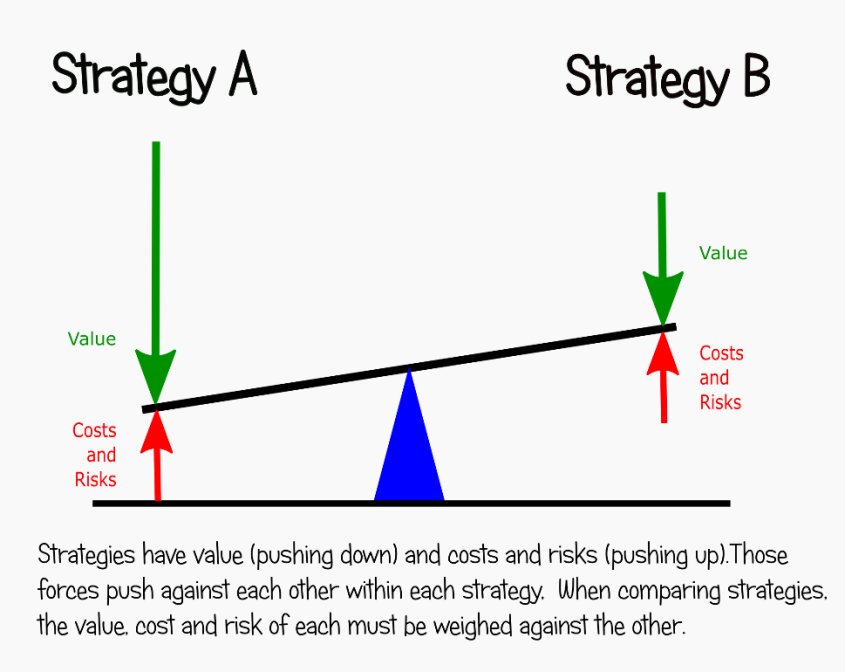


Figure 3

What we do with those understandings ultimately results in mental calculations and decisions along the lines of Figure 3. In context-driven work, our choices are guided not according to a fixed script of "best practices" but rather by dynamically evaluating context and selecting, designing, or adjusting our actions to solve the problems that we encounter. We don't simply look at whether a particular strategy is worth doing in and of itself, such as strategy B in the diagram, where you can see that its value outweighs the risks and the costs. We also compare that to other strategies that might be even better, such as strategy A.

Yes, these decisions may be biased, as in Figure 4, perhaps because we unconsciously veer toward things we know and away from potentially wonderful new ideas that we aren't yet comfortable with. But still, we strive to make decisions based on merits rather than following the dictates of fashion or arguments from authority. In context-driven testing, we don't idolize "best practices."

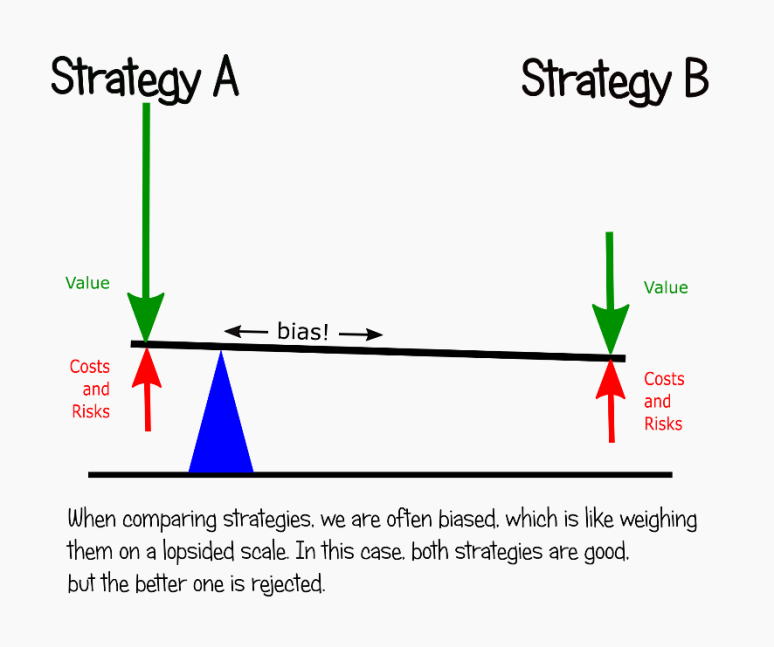


Figure 4

The answer to the question of how context drives tooling is: we read the situation around us; we discover the factors that matter; we generate options; we weigh our options, and we build a defensible case for choosing a

particular option over all others. Then we put that option into practice and take responsibility for what happens next. All along we are learning and getting better at this.

Building a test strategy and determining how to use tools to fulfill that strategy is an evolutionary process. No one who says "do it right the first time" has ever really done anything difficult right the first time. We become able to do things well partly via the experience of doing them badly. We often learn how to develop powerful, polished tools by developing cheap, disposable tools—and then throwing them away and applying what we've learned. Developing software also means developing our approaches to testing it.

Context-driven behavior tends to be highly exploratory because the practitioner is responsible, at every turn, for the quality of the work—and not just the immediate work, but also the overall strategy. If you are not just following instructions handed down from a boss or bureaucrat, then you have to evaluate the situation and frequently adjust your practices to get the best result you can. This also means that Context-Driven practitioners think not only about efficiencies, but about contingencies as well.

Approaching test tooling in a context-driven way means we don't play down the problems that tools have. We try to face them forthrightly. This can make us look rather pessimistic about certain ways of using tools, though, so it is a special challenge to remind ourselves of the benefits of tooling and to move toward the kind of tools that provide more of those benefits.

Do We Reinvent Every Wheel?

In practice, very little we do is designed from scratch. We collect and apply reusable heuristics by which we quickly solve common problems. We don't call these "best practices," because they aren't. They are patterns we find useful in particular situations, and we apply them mindfully. This involves looking to the cost, value, and contingencies of any given heuristic; and it involves ongoing re-evaluation of our process.

Invest in tools that give you more freedom in more situations.

So, how does a Context-Driven tester approach tools and their use? Well, there are no "best tools" in the Context-Driven world. Indeed, there are no dedicated "test tools" in the Context-Driven world. Any tool can be a test tool. Any tool might be useful. But we can suggest, all other things being equal, factors that make some tools more generally preferable. In good Context-Driven fashion, we acknowledge at least one exception for each heuristic:

1. **Tools that support many purposes are preferable to those optimized for one purpose.** Some tools are designed with specific process assumptions in mind. If you work in a context where those assumptions apply, you are fine. But what if you decide to change the process? Will your tools let you change? In changing contexts, tools that are simple, modular, or

adaptable tend to be better investments. Tools that operate through widely used interfaces and support widely used file formats are more easily adapted to new uses. Note that a tool may have only one major function, such as searching for patterns in text, and yet be a good fit for many purposes and many processes. *Exception:* If a tool happens to fulfill its one purpose far better than alternative tools, it might be worth the trouble of making room for it in your toolbox.

2. **Tools that are inexpensive (or free) are preferable to expensive tools even in many cases where the expensive tools are more powerful.** This is partly because of "sunk cost bias." The more money management pays to acquire a tool, the less acceptable it is to stop using the tool even if the tool is obviously unsuited for the purpose at hand. Furthermore, free tools invite us to experiment with different techniques. Experimenting is absolutely necessary in order to develop the skills and knowledge we need to make informed decisions about our processes. *Exceptions:* Remember there is more to cost than the purchase price. An apparently inexpensive tool may cost more in the long run if it requires extraordinary maintenance. Also, an expensive tool might be the only tool that has the special capabilities that you seek.
3. **Tools that require more human engagement and control are preferable to those that require less.** This is due to a syndrome called "automation complacency," which is the tendency of human operators to lose their skills over time when using a tool that renders skill unnecessary *under normal circumstances*. In order to retain our wits, we humans must exercise them. Tools should be designed with that in mind, or else when the tool fails, the human operator will not be prepared to react[6]. *Exception:* We may genuinely value the power and convenience that the tool gives us more than we value the skills and awareness that we lose in the process.
4. **Tools that are supported by a large and active community are preferable to those that are not.** The more people who use a tool, the more free support will be available and the more libraries, plug-ins, or other extensions of that tool. This increases the value of the investment in learning that tool, while reducing the learning curve. (The R language is a good example. It's a powerful and general purpose data analysis tool. Lots of researchers use R, lots of books about it are on Amazon.com, and there are hundreds of libraries that provide special capabilities beyond the defaults capabilities of the tool.) *Exception:* Just as in the case of expensive tools, sometimes the value you get from a tool is so important that it overrides concerns about support.
5. **Tools that can be useful to non-specialists are preferable to those that are not.** We're talking about tools that lower the cost of getting started; that afford ease of use; that don't depend on proprietary languages; that have lower transfer and training cost. Microsoft Excel and spreadsheets in general provide a good example. It is possible to use Excel in a very

[6] See Nicholas Carr, *The Glass Cage,* and Lisanne Bainbridge, "Ironies of Automation", *Automatica*, Vol. 19, No. 6. pp. 775 779, 1983.

specialized and sophisticated way, but there is a lot Excel can do for you, even if you have only basic skills with it. *Exception:* Sometimes it can be good for a tool to dissuade non-specialists from using it, because non-specialists may not be capable of using the tool wisely.

6. **Tools over which we have control are preferable to those controlled by others.** Good tools are at the very least configurable to your specific needs. An open source tool allows you to control every aspect of it, if you need to do that. Apart from the expense, commercial proprietary tools prevent you from adding new features and fixing critical bugs. Proprietary tools may be modified in ways that disrupt your work at inconvenient times, and you have no control over that schedule. *Exception:* Sometimes not having control over a tool is a good thing, because you are forced to use standard versions and configurations which allow you to share work more easily with others who use that tool.

7. **Tools that are portable across many platforms are preferable to those restricted to a single platform.** One aspect of context is the operating system or hardware platform. Cross-platform tools obviously work in a wider context. *Exception:* A tool may provide value that is important enough to offset its lack of cross-platform compatibility; or it may offer interoperability with similar tools on those other platforms.
8. **Tools that are widely (or easily) deployed are preferable to tools that aren't.** A primary problem of tool use is getting the tool in the first place. Some tools require complicated installation and configuration. Some tools may require special permission or expenditure. This can require negotiating with the IT department, managers, or co-workers. *Exception:* Some tools may be worth this trouble.

Invest in testability[7].

The success of any tooling strategy depends in large part on how well your technology affords interactions with tools. This is why it pays to build testability into your products. From a tool perspective this means two big things: that your product is controllable by tools via easily scripted interfaces, and that its states and outputs are observable by those tools. For this reason, browser based products tend to be more testable while apps on mobile devices are less testable. Products with standard controls are more testable than products with custom controls.

[7] See http://www.satisfice.com/tools/testability.pdf

Consider making testability review a part of each iteration, and otherwise instill this thought process early in your projects. Any serious attempt to make tooling work must go hand-in-hand with testable engineering.

Let's see tool-supported testing in action!

We now present you with three examples of how tools help testing. The first example involves no checking. The second is checking done partly with tools and partly by the tester. The third is fully automated checking through the GUI (and gives you an idea of why we generally avoid doing that).

These examples are worked from the perspective of the independent tester, rather than the developer. We will demonstrate by describing some of our testing of FocusWriter, a word processor with a minimalist design, intended to create a distraction-free environment that helps authors write novels.

CASE #1: Tool use without checking.

> *James wanted to test FocusWriter. He opened his browser, navigated to the gottcode.org Web site, downloaded FocusWriter, extracted it from its .zip file, and played with it.*

Are there any tools in use here, so far? Most testers would say no. But seemingly, according to our definition, the computer is a tool of some kind. Various parts of the computer are tools, such as the mouse, the monitor, and keyboard; hardware. The web browser James used to download the product is a software tool. The website he accessed is a tool, too. The Internet itself is a tool. Yet, no one feels that these are "testing tools", nor that this activity is anything like "test automation."

Why? Because none of these things are tools with respect to anything unique about testing. Instead, they comprise the fabric of ordinary computing; ordinary use of the product. When we speak of tools in testing, we do not mean the natural processes of using the product, but rather contrivances applied for the purpose of accelerating or enabling testing over and above ordinary human interaction with the product.

James opened SnapTimer and set a 15-minute rolling timer. He opened Evernote and started a new note titled "FocusWriter Test Session." Then he Googled FocusWriter and thumbed through the top hits. In this way, he discovered that FocusWriter is an open source app.

The tools here are SnapTimer, Evernote, and Google. These are not part of the FocusWriter user experience, and they are not employed in simulation of what a user would do in the ordinary business of using FocusWriter. Therefore, these are bona fide test tools in this case. They are applied specifically for testing purposes, even though they may not have been designed for testing per se.

More tool use quickly followed:

1. *James located the source code on Github. (Google)*
2. *He used Git to download that code. (Git)*
3. *He unzipped it. (7Zip)*
4. *He opened a Windows command prompt. (CMD)*
5. *From the top of the source directory he used* grep *to search for the string "error." (grep)*
6. *He noticed this line in the output:* ***m_error = tr("Unable to open archive.");*** *On the conjecture that "tr" means translate, and therefore may be the formal mechanism for displaying localized message strings, he used a regular expression search to extract every associated string from the source. (grep with regex)*
7. *He extracted the strings themselves using this Perl program:* ***while(<>) { foreach (/tr\("(.*?)"\)/g) { print "$_\n" } }*** *(Perl with regex)*
8. *He used Notepad++ TextFX plugin to sort the result and eliminate duplicates. (Notepad++ with TextFX)*
9. *He grouped all commands together, all keyboard shortcuts together, all messages to the user together. (Notepad++)*
10. *At some point during this process, the 15-minute timer chimed, which alerted James to the need to update his notes and check in on his test charter. (SnapTimer and Evernote)*

Strings extracted from source code (as of step 9):

```
Left
Line Spacing
List all documents
Loading settings
Loading sounds
Loading themes
Longest streak
Manage Sessions
Margin:
Memo:
Minimum progress for
streaks:
Minutes:
Misspelled:
...
The requested session
name is already in use.
Unable to load
typewriter sounds.
Unable to open archive.
Unable to overwrite
'%1'.
Unable to rename '%1'.
Unable to save '%1'.
Unexpectedly reached end
of file.
...
&About
&Add
&Bold
&Change
&Close
```

We see plenty of tool use here, but most people would not call this "test automation." So what? It is tool-supported testing. Testers should think about tools as helping them in *any* aspect of testing.

This case also shows the power of a tester who already knows how to use basic, free, technical tools and possesses a foundation of technical knowledge sufficient to read and write code. Not all testers need that—but we suggest all testers need *access* to someone who does have that skill, such as a toolsmith on the team.

The use of tools in this case led to interesting results:

- Discovery of functionality referred to inside the product ("&Daily Progress") but not yet implemented.
- Discovery of error messages relating to previously unknown functionality.
- The basis for systematic testing of error handling.

CASE #2: Tool-support via patterned data generation for better coverage and a powerful oracle.

James frequently uses tools to generate special test data. One of his favorite tactics is called a "simplified data oracle." For FocusWriter he used that to test the Scene List and Filter functions.

The Scene List is a feature that allows the author to navigate, select, and move scenes more easily. A scene is a block of text delimited by a specific text marker, such as "##."

To test the Scene List we need a document that has scenes in it. That's easy. We create some text and put some scene dividers into it, as in Figure 5. This text will display in the Scene List as in Figure 6. Let's say this looks good. Let's say it is exactly what we wanted to see.

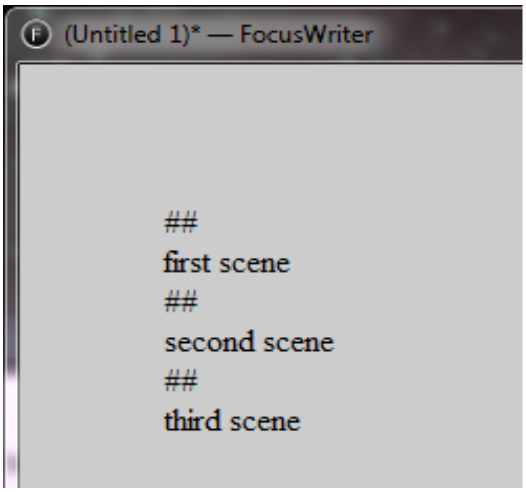


Figure 5

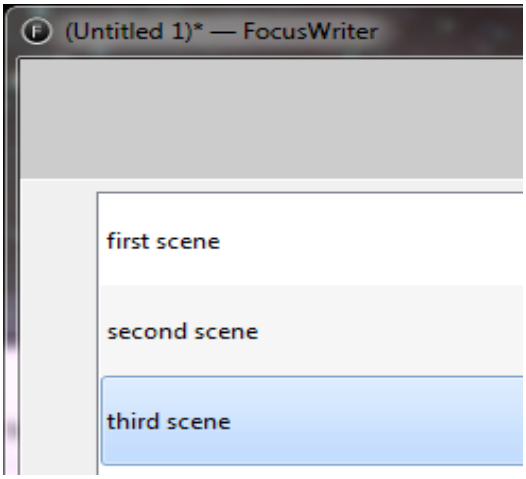


Figure 6

Now we could automate this as a typical output check. But tools can do so much more for us. So, James decided to write a program that would create thousands of scenes, and identify them in a specific way that would help us track whether they were in the correct order in the Scene List. In other words, it is a combination of a stress test and a correctness check (to be performed by a human tester) that helps us see if there is a bug in how the Scene List displays and sequences the scenes. It helps us test the scene divider handling for a variety of different divider strings (because the scene divider string is configurable) as well as scene filtering functionality.

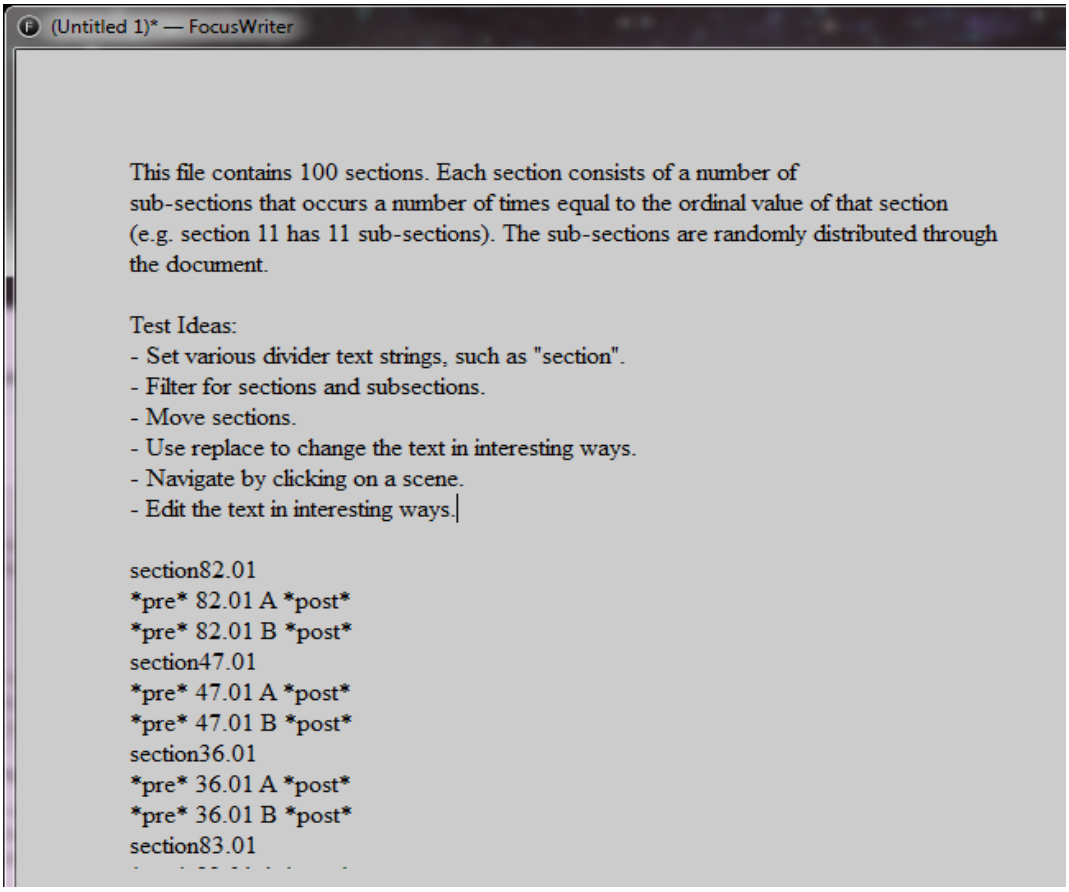


Figure 5

The program he wrote results in the file shown in Figure 7, which contains a total of 5,050 sub-sections. If the scene divider string is set to “scene” then the Scene List shows Figure 8.

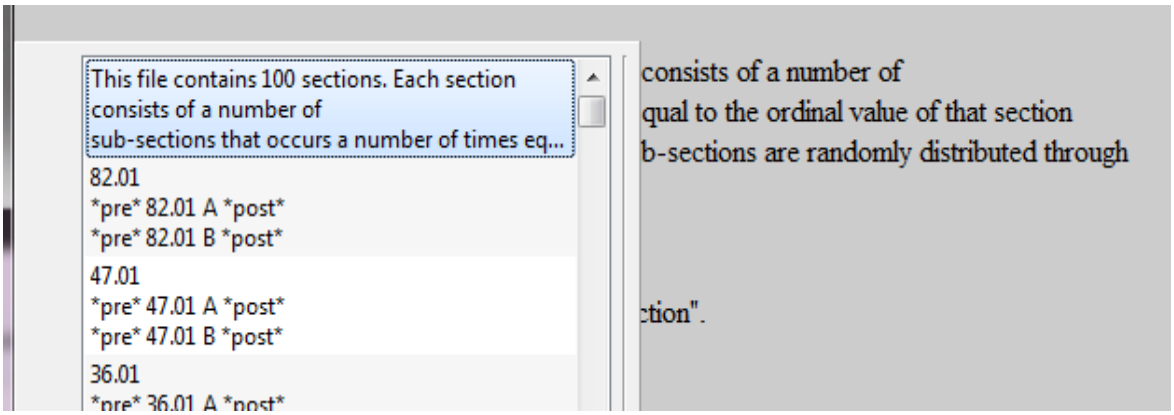


Figure 6

Now, by filtering on “07.” it should pick out only seven scenes, wherever they are in the document, and display them in order in the Scene List panel. This is in fact what happens.

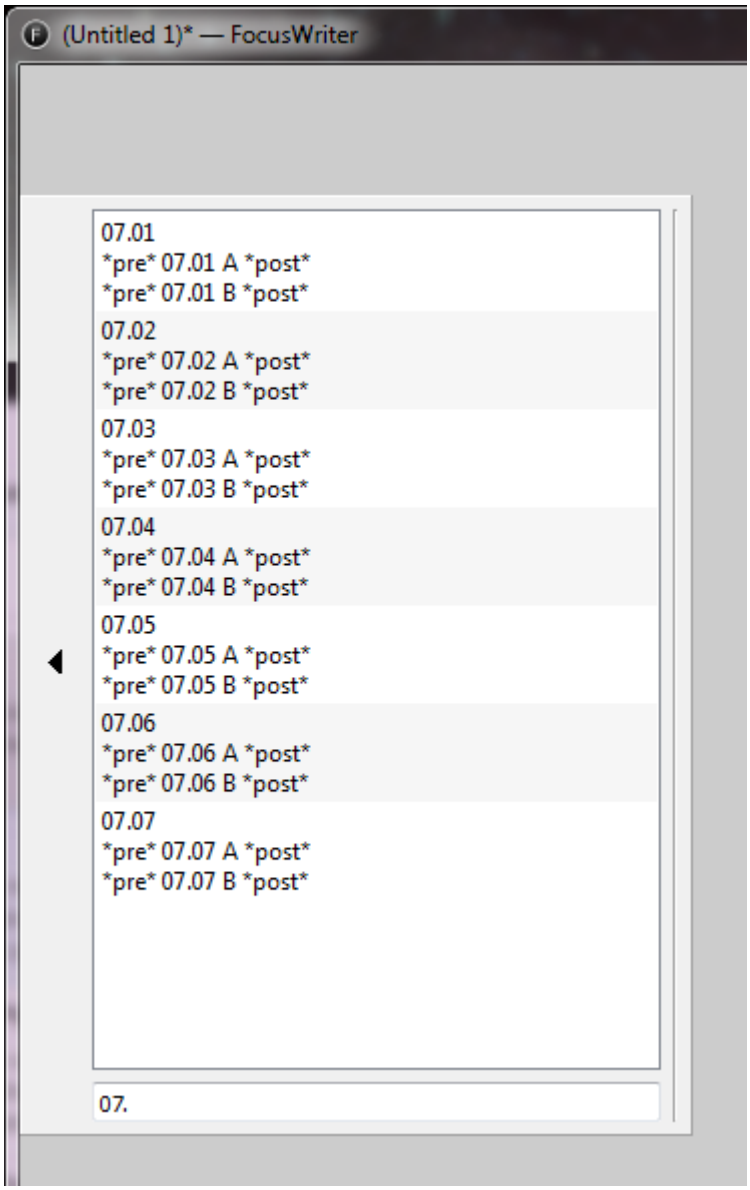


Figure 7

This is testing with a big boost from a tool. The tester can play. The tester can move scenes, filter, edit the document, or whatever. All the while, his test coverage will be deeper and his oracles sharper because of this patterned data. The tester is not limited to using the data, but can also edit the program that created the data to create even more interesting data. In fact, the version of the data you see, above, is the fifth refinement of the original concept.

CASE #3: Automated checking.
We wanted to demonstrate full-on automated checking, while at the same time staying with the FocusWriter example. FocusWriter does not provide an API for testing, which required us to automate through the GUI, and so the headaches began.

The high concept for the check was straightforward: perform a series of operations in FocusWriter that touch several features, change a document in a few ways, but result in the same output as we had at the beginning. A check that ends with the same state it began with is called *idempotent.* Idempotency is a useful heuristic because the process should be repeatable any number of times without regard for any progressive issues with system state—and if the state of the system interferes with the automation, that would be an interesting test result.

James proposed a process to be automated:

1. Delete any old temporary test files.
2. Start FocusWriter.
3. Load the Three Musketeers text file.
4. Search and replace all instances of lower case "e" with "~" (chosen since it does not appear in the text).
5. Save the file as type ODT.
6. Close the file.
7. Close FocusWriter.
8. Open FocusWriter.
9. Open .ODT file.
10. Search and replace all instances of "~" with lower case "e."
11. Save as TXT in new file.
12. Compare original with new text document.
13. Log result.
14. Exit FocusWriter.

This is designed to exercise saving (two kinds of files), loading (two kinds of files), starting, stopping, searching, and replacing. It comprises a bit of a stress test, because of the size of the Three Musketeers novel (1.3 million characters), but mainly it would be useful as a sanity check.

At James' suggestion, Michael started to tackle the task using AutoHotKey, a Visual-Basic-like Windows scripting language. He soon ran into a problem: he was unable to query and confirm the state of the list box control by which the user chooses the file type. That obstacle and his unfamiliarity with AutoHotKey prompted him to switch to Ruby, with which he has a good deal

What about the unit level?

Automated, low-level checking is most famously embodied by the practice of "test-driven design" (which is really "output-driven design" but now it's too late to rename it). We are not going to cover it here because it's too big a subject and already gets so much coverage in the Agile world.

Automating low-level checks is a powerful practice that can improve testability and make quality easier to achieve. Like all checking it requires skill and forethought to pull off, and it is blind to many bugs that occur only in a fully integrated and deployed system. Still, it is generally much less trouble and expense than GUI-level checking.

more experience. Ruby has several libraries that provide support for the Windows API. He soon found the RAutomation library which is billed as "a small and easy to use library for helping out to automate windows and their controls for automated testing". Like many such open-source offerings, it is sparsely documented, but with a few minutes of experimentation, Michael was confident that he would be able to make sense of it and put it to work.

RAutomation proved to be intuitive and straightforward to use, but Michael quickly discovered that certain aspects of FocusWriter made automating the process tricky. Among other things, FocusWriter appeared to implement list boxes such that RAutomation (as AutoHotKey before it) could not determine the currently selected option for the current file type; Michael had to track this by other means. Saving a file would sometimes cause a confirmation dialog to appear, sometimes not. Several dialogs shared the same caption ("Question"), even though the prompts and options within were different. Frequently the script would initialize actions before the application was ready for them, requiring wait states of one kind or another. All of this required loops of experimentation, discovery, learning, and revision that, in the end, took hours. Among other things, Michael wished that he had been part of the development process for FocusWriter to appeal for better testability.

After much fiddling Michael succeeded in getting the process running reliably, but when James used the same script on his own system, it was not able to find and start FocusWriter! After an hour of investigating together, we abandoned Ruby.

We considered our problems so far. Perhaps what we needed was a tool optimized for interacting with the product via the GUI. We've heard of HP Unified Functional Testing and its predecessors from testers forever. HP says *"HP UFT software automates testing through an intuitive, visual user experience that ties manual, automated, and framework-based testing together in one IDE. This far-reaching solution significantly reduces the cost and complexity of the functional testing process while driving continuous quality."*[8] To test this claim, we downloaded the trial version. After another full hour using the record and playback facility of HP UFT, we were able to get FocusWriter started, but could not get HP UFT to recognize the application window. It recognized Notepad, but there seemed to be something about FocusWriter (the fact that it is built with the QT toolkit?) that made it invisible to the HP tool. HP UFT would record a script, but then was not able to run its own script! We changed settings and edited the script in different ways, all to no avail.

Perhaps another five minutes or five hours would have gotten us past the problems with HP UFT. Neither of us are expert in the use of this *particular* GUI automation tool, and some experience with it might help us get around some of the obstacles. Perhaps our programming skills would accelerate our learning curve. Yet tools like this are often marketed in terms of "test automation without programming skills". Here's a typical example: "Test automation alleviates testers'

[8] http://bit.ly/1j9VUCL, retrieved October 30, 2015.

frustrations and allows the test execution without user interaction while guaranteeing repeatability and accuracy." Claims like these were being made 20 years ago. Meanwhile, the self-driving flying cars are still very much on the ground, with human drivers behind the wheel.

As a final effort, James used AutoHotkey to record a macro that performed the basic script. Success!

We found bugs…but not because we automated the check.

During this experiment in checking, we found that the final TXT file did not match the original one. That is all that checking can do: report some sort of inconsistency that must then be investigated.

Our subsequent investigation of the inconsistency led to two bugs:

1. FocusWriter writes ODT files in a manner that Microsoft Word complains is invalid (although it is apparently able to rescue the content).
2. FocusWriter reads ODT files incorrectly, causing one extra line and ten new spaces to be inserted wherever there is a line that begins with at least two leading spaces.

We reached our understanding of the first problem because we thought to use Microsoft Word and OpenOffice as "comparable product" oracles for the evaluation of the ODT file saved by FocusWriter. We reached our understanding of the second problem using a bevy of tools:

- **WinMerge** to analyze the text differences between the files
- **Frhedit** to analyze the hexadecimal differences between the files
- **Perl** to create and modify test files with various properties in order to test our hypotheses
- **7Zip** and **Notepad++** to examine the XML content of ODT files
- **Excel** to build a spreadsheet that predicted size changes
- **Wikipedia/Google** to study UTF-8 encoding

Note that automating the check had little to do with finding and investigating these bugs. It didn't save us any time. In this case, so far, the automated check is a cost without benefit. The check itself in its un-automated form—specified by a thinking tester, and carried out interactively during the process of automating it—simply gave us an indication of a problem. Then, skilled testers carried through with the investigation (with the help of tools) that systematically pared down potential factors to home in on the few that mattered.

The foray into automated checking took far more time than the first two cases. We learned things in the process of developing this automated check that might have made further checking easier to some degree, especially if our experience could inform better testability. Yet we wonder: if we were to create a library of checks, would the value of such checks match the development cost? The maintenance cost? The opportunity cost?

It may be that, sometime in the future, another bug will creep in to the product that this particular check will notice. If there are changes to this area of the product in the future, they will probably cause our check to fail irrespective of whether the failure is due to a bug in the product or in the script. If there are no changes in this area, the check will not fail, but probably also will not be worth running. It is often difficult, ahead of time, to choose which checks will be worth having automated, and which will turn out to have been white elephants. A lot of this is a matter of guessing about risk and change. This involves skill, experimentation and learning.

Why is automating interactions through a GUI so difficult?

After hearing this story, some "test automators" might claim that they don't have these kinds of problems, and if they did, they would be able to get around the problems easily. In saying so, they would be completely missing several points.

GUIs are designed for able-bodied humans, who don't have the same trouble finding and interacting with products as robots do. In fact, our tools fell victim to the same kinds of barriers presented to people who suffer from disabilities and yet try to use modern technology. Accessibility is a widespread problem in computing[9] for people and tools alike. Just as success with one accessible product does not disprove the existence of the accessibility problem in general, pointing to one trouble-free use of a tool to control an app doesn't disprove our general claim, which is this:

GUI-level scripting is fiddly. It's fussy. It fails suddenly in unexpected ways. It's notoriously problematic. It requires learning not only about the application and the tool, but also about how they will interact. We both learned this in the 80's and 90's[10], we've seen it ever since, and we are seeing it now.

What Every Toolsmith Knows

• • •

"GUI-level scripting is fiddly. It's fussy."

Beyond the accessibility issue, think about it. Even something as simple as saving a file, which is easily navigated by a human, becomes an exercise in pure pedantics when you attempt to program a machine to do the same thing. When you save a file, the application **may or may not** present dialogs that are distinguishable using a particular approach; it **may or may not** use libraries or controls that are recognized by the test framework you're using. On any given run, the application **may or may not** be pointed to the right directory; it **may or may not** ask you if you really want to overwrite another file; it **may or may not** refuse to proceed because that other file is locked by another process or because disk space ran out on your USB stick. The application **may or may not**

[9] http://arc.applause.com/2015/11/02/mobile-accessibility-end-user
[10] See James Bach, "Test Automation Snake Oil", http://www.satisfice.com/articles/test_automation_snake_oil.pdf

be interrupted by some other application during this process. It **may or may not** take an unusually long time to save.

Humans take all these eventualities in stride—we barely notice them unless they seem wrong! Not so for programs. Any possibility that has not been anticipated and not expressly programmed is a potential stumble for the script, which means another round of troubleshooting, debugging, and testing for the person trying to program it. Even if you think you could do better with FocusWriter using your favorite tool, we claim that the *kinds of problems* we have highlighted are not unusual in tooling generally, and that they are *normal* to GUI-level user simulation tooling, regardless of your skill level, industry, or product type.

Although Michael succeeded with Ruby and James succeeded with AutoHotKey, without more work, success took the form of rough prototype scripts. These are very brittle. The smallest change in the application, or in the saved state of the application, or in the data set may disrupt the script in a way that requires tuning or a complete rewrite.

You can make GUI checking more resilient in the face of product change, at a price…

As one of our reviewers, Ben Simo, noted, we can certainly make GUI checks less brittle in the face of change, but the very factors that make them less brittle usually make them less powerful, too, because we achieve greater resiliency by sacrificing certain sensitivities. For instance, we may filter out time stamps or user names, or we block out sections of screenshots. It may be okay to ignore the current user name when we are running the tests with accounts called "TestRobot6" or "TestRobot22" and want to use the same logic to check screens in both cases, but what if there is a legitimate bug whereby the user's name is wrongly displayed? Our modified check won't spot it. Adding such special case logic into the check code also increases the complexity of that code, which creates brittleness of a different kind: an increased likelihood that we will break the code when we try to improve it.

We may be completely successful in suppressing certain disruptions to GUI automation, but those same disruptions may give us information about the product that, as human users, we would easily understand as significant. For instance, we can implement a time delay in checking output from a process in order to assure that the process is complete before we attempt to process the output, but that means we won't notice if it gets progressively slower and faster in its response times *within* that time delay. Testers don't just numbly watch the world go by. Real testers are not just idle product tourists—we critically analyze what we see. But if we outsource our "seeing" to the computer, we cannot be critical of what *it* sees, and the computer doesn't know *how* to be critical.

As you navigate these troubles, you will probably be caught up in a more insidious pattern: GUI-level checking distracts testers from performing deep tests that examine complicated or subtle functional behaviors. This is because you have to keep the checking simple. If you pour complex, interesting data and interactions into your checks, you will create huge headaches for yourself in

coding and maintaining it. Imagine reproducing, in code form, just five minutes of your typical use of your computer. Yikes. This is why GUI check designers focus on superficial interactions and easily parsed outputs. It's economically viable. It will allow you to add more "test cases" to your "test suite", but what it accomplishes is likely to be shallow testing. At the same time, all this effort presents opportunity cost that robs you of time for deeper testing.

There are contexts in which automated checking is likely to be cheaper and more powerful. Products that are built with testability in mind—scriptable interfaces and log files that can be easily parsed—tend to be more amenable to automated checking. Products that have simpler forms of input and output are easier to check programmatically. Automated checks closer to the developer's current task can afford quick change detection, fast feedback, and simpler repair. Well-built, "unbuggy" products can be much easier to automate and to check.

These points we are making are not new. The first Los Altos Workshop on Software Testing, which was the first organized gathering of the not-yet-named Context-Driven School of software testing dealt with the problems of automating the interaction of an application through a GUI—and that was way back in late 90's[11].

Automating actions is a tactic. It should not be a ritual.

In the Context-Driven world, we reject ritual. We embrace problem-solving. But this attitude is only valid if problem-solving matters. Too often, automation (in both senses—artifacts and enterprise) is pursued as an unquestioned good; stuff that dazzles people even when it accomplishes little. Large tool companies and consulting firms aren't much help, either: it's not in their interest to help you see a simpler, cheaper, more flexible way of doing things.

If your testing doesn't really matter, except as a display for public relations purposes, then maybe rituals are acceptable— but that cannot be so if your intention is to find important bugs before it is too late. To fulfill *that* mission, you must develop an appreciation of the full spectrum of tools and their applications to your work. Context-Driven testers apply tools in powerful ways to get testing done!

[11] Kaner, Cem, *Improving the Maintainability of Automated Test Suites*, p. 10, http://www.kaner.com/pdfs/autosqa.pdf

TAIT ELECTRONICS

Rapid Testing Clinic

February 15th and 16th, 2007

Event Overview

Six testers were assigned to test a new product feature using rapid testing methodology, under the observation of James Bach. The testers were Andrew Robins, Josh Crompton, Sridhar "Raj" Kasibatla, Matt Campbell, Dan Manton, and Judy Zhou. The product feature was dual-head radios. The event was held in a dedicated conference room. Lunch was brought in, which allowed us two full uninterrupted work days.

Andrew Robins served as overall test lead. James Bach served as commentator, facilitator, and part-time scribe. Each of the testers had taken the Satisfice Rapid Software Testing class. The testers were chosen for their testing skills and leadership qualities.

The goals of the clinic were 1) evaluate and coach the testers in their understanding and application of the Satisfice Rapid Testing methodology, 2) give the testers experience in how a testing clinic might be run, 3) test dual-head radios, and 4) prepare to test dual-head radios better going forward.

Process Overview

This is how the event unfolded:

Day 1

1. **Set up equipment and room for testing.** This was done the night before the event began.
2. **Outlined and discussed new, changed, and risky areas of the product.** See whiteboard snapshot and mindmap transcript.
3. **Created reference diagram of dual-head radios.** This served as a working reference for test strategy and chartering.
4. **Split into three testing pairs and set the charter for the first session.** Everyone's charter for the first session was the same: recon. Set up and look over dual-head functionality in action, learn how it works, observe it in action, get to know the risk areas first hand.

5. **Performed 90 minute test session.** As expected much of the first session was spent attempting to get the equipment properly set up. The bug database was checked several times to compare observations with known bugs. Basic misunderstandings about how dual-head radios work, such as how databases are updated and used, were revealed. This is characteristic of "high learning" exploratory testing. James filmed some of that process.
6. **Debriefed the sessions together.** Bugs and setup problems were discussed. We also discussed how session-based productivity metrics work, and the teams estimated their "TBS" proportions as well as "on charter" test percentages. James explained how debriefings worked and answered some questions. We also called Jeremy, the analyst, into the room to answer questions about a mysterious feature of the product. In the end, we noted the mystery but Andrew decided it was not important enough to pursue at that time.
7. **Performed 60 minute session.** We continued on the same charters ("recon sessions"). As expected, there was more testing and less setup in the second session.
8. **Debriefed the sessions together.** Two teams felt like they had reached a plateau and were ready to move on from recon work.
9. **Set up an overnight stress test.** Josh scripted something quick and dirty to keep the heads going all night.
10. **Worked up combination tests (evening).** James wrote software to demonstrate combinatorial state transition testing.

Day 2

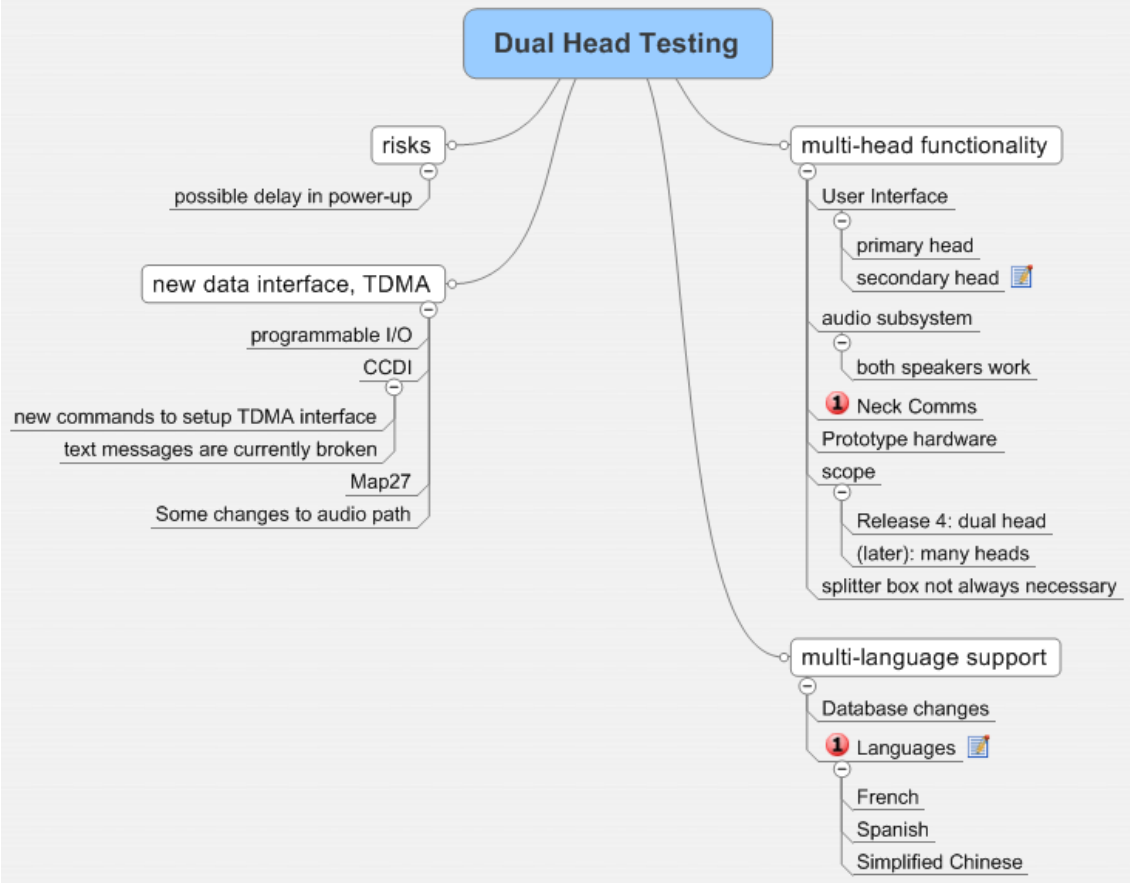
11. **Guided brainstorm of test ideas and product elements using the Satisfice Heuristic Test Strategy Model.** This began with a silent 10 minute review, followed by 90-minute discussion and elaboration of ideas.
12. **Performed 75 minute "analysis and coverage" test session.** This session was focused on understanding a specific sub-feature or risk better.
13. **Debriefed the sessions together.** Bugs were discussed. We called in a firmware programmer to help us understand database synchronization issues.
14. **Made master list of bugs found.**
15. **Performed 75 minute final test session.** One team tried some of James' generated state transition tests. Another team worked up a table to bring more structured coverage to the brown-out tests.
16. **Final debriefing with Heiko and bug list update.**

Artifacts

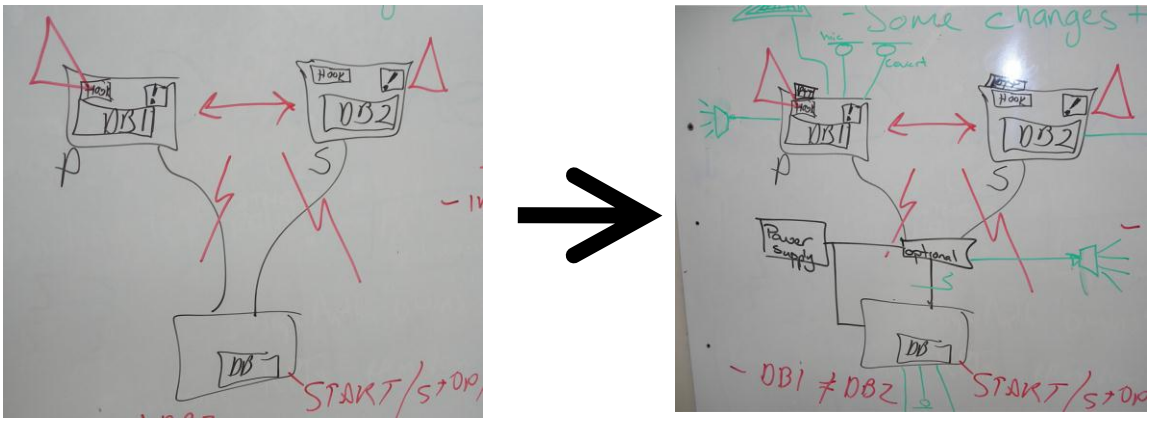
Various pictures and videos of the process can also be found accompanying this document.

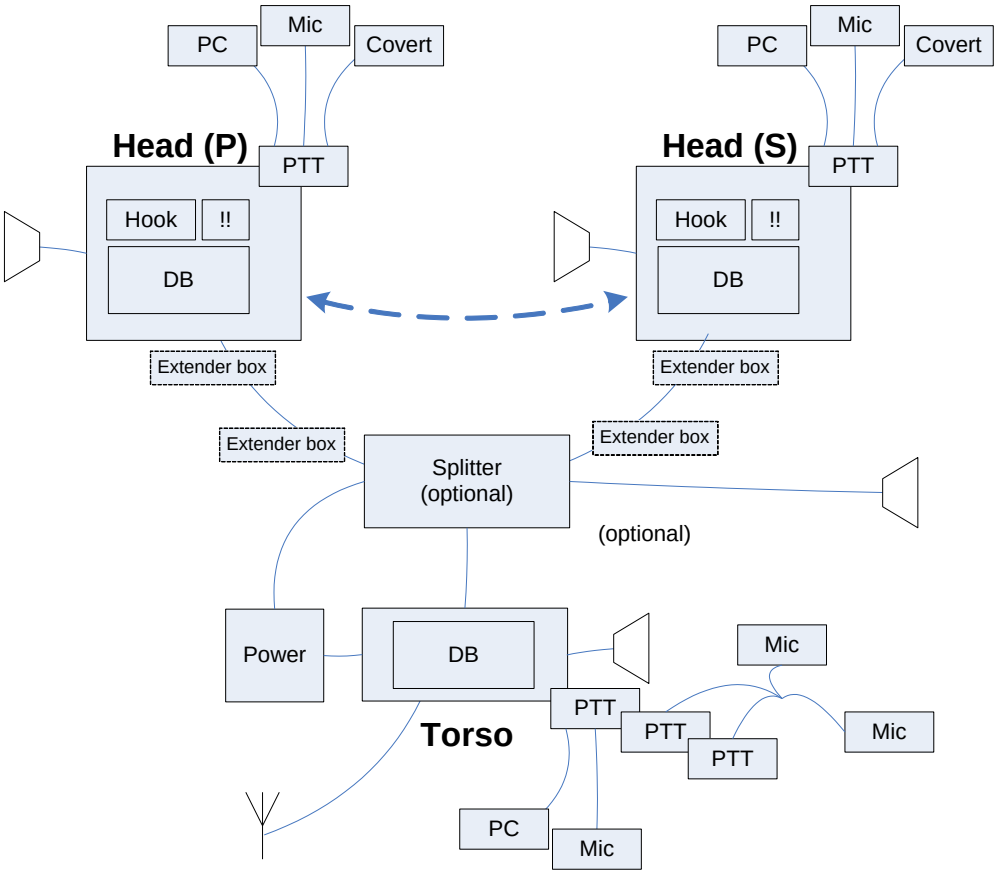
Day 1: Briefing on Changed/Risky Areas

Transcribed from the whiteboard, it was decided not to worry about language support testing. A lot of testing was to be focused on neck comms, audio sub-system, primary/secondary interactions. Also looked at delays and brown-outs.



Day 1: Dual-Head Diagram Evolution





Coverage

- DB != DB, DB == DB
- Disconnect/Connect
- Start/Stop/Restart/Reset
- On hook/off hook
- PTT Y/N
- Signal arriving at antenna
- No testing for extender box?!

Oracles

- Screen Match
- Contrast/Volume Independence
- Muted/Unmuted
- Reset on Disconnect
- Reset on System Error
- Pops

Ideas

- Happy path
- Spam test
- Connection tests (failover)
- DB interactions
- Pairwise interactions
- Head interactions
- Time (leave it sitting)

Day 2: Product Element And Test Idea Brainstorm (lightface: categories, boldface: ideas)

Test Techniques
   Function Testing
   Domain Testing
   Stress Testing
Soak testing

   Flow Testing
   Scenario Testing
   Claims Testing
   User Testing
   Risk Testing
   Automatic Testing
Project Environment
   Customers
   Information
   Developer Relations
   Test Team
   Equipment & Tools
   Schedule
   Test Items
   Deliverables
Product Elements
   Structure
Some units click/pop and others don't
      Code
Software Feature Enabler
??? upgrade path for firmware? How do you upgrade it?
??? upgrading in situ? (on vehicle)
??? what if different heads have different firmware/hardware versions?

      Interfaces
??? Audio feedback issues with multi-head?

      Hardware
Splitter box
   ??? reduce audio quality?
Keypad microphone serial protocol
GPIO protocol
Remote kit variants
Different Band Torsos (sanity check)
   UHF
   VHF
Microphones
   Keypad
   Standard
   Dynamic
Control head variants
Neck Comms
   Cable length
   Timing issues
   Connections
   Brown out issues
   RF interference on cables

      Non-executable files
      Collateral
Installation guide for dual heads
??? ability remove the splitter box and connect directly

Functions
User Interface

Lockout between heads
Interruption of some kind during lockout
(e.g. disconnection event, emergency mode)

System Interface
Serial communication via secondary head by mistake

Application
Autodetect of microphones
Software functions enabled/disabled
Software function enabling
??? Thermal shutdown? Thermal senors?
Security lock
Dual head menu
Hook switch listen-in mute
Listen in

Scanning mode vs. non-scanning mode

Calculation
Time-related
Audio lag from torso to the different heads
Transmit inhibit
Auto-power-down
Transformations
Database changes

Startup/Shutdown
Disconnecting and connecting a head (primary vs. secondary)
System reset
Power disconnect
Normal start
Normal shutdown
(??? Any way to start via system interface?)
??? Soft disconnect of a head? In situ, you may not be able to unplug it.
??? What if a head has malfunctioned in a way that denies service, and you need to sever the connection?

Partial powerup scenario—can we get the head to power up without torso? One head powered, another not?

Multimedia
Exact bits on screen
Pops and clicks

Error Handling
(??? Error messages? Can we get a list?)

Interactions
Testability
Data
Input

Editing scan group on one head, another head interferes (power off?)

Programming the radio, disconnect head before committing.

Output
Preset

(??? Configurable menu items that may relate to interactions between heads?)

Persistent

Database integrity on the heads (how do you know that the database is what it should be?)

Contention between databases in each head

Legitimate/common ways that databases could be different between two heads.

Same mac address/priority?
Sequences
Big and little
Very large scan groups
Large numbers of channels
Lots of data for MPT which is redundant anyway

Noise
Rattling

Lifecycle
Initial configuration of database, plus sequences of changes, then exporting and importing to another radio

Platform
External Hardware
GPS data display on heads

External Software
Internal Components
Operations
Users
Environment
Common Use
Disfavored Use
Extreme Use
Time
Input/Output
Fast/Slow
Changing Rates
Concurrency
Quality Criteria Categories
Capability
Reliability

Memory leak issues?

Error handling
Data Integrity
Safety
Usability
Learnability
Operability
Accessibility
Security

```
      Authentication
      Authorization
      Privacy
      Security holes
   Scalability
   Performance
   Installability
      System requirements
      Configuration
      Uninstallation
      Upgrades
   Compatibility
      Application Compatibility
      Operating System Compatibility
      Hardware Compatibility
      Backward Compatibility
      Resource Usage
   Supportability
   Testability

Diagnostics?

   Maintainability
   Portability
   Localizability
      Regulations
      Language
      Money
      Social or cultural differences
```

Day 2: Bugs Found

Andrew/Josh

Day 1

- Can't disconnect heads without special tool
- Problems with minimum volume
- Audible artifacts associated with keypresses on control heads (press and release)
- PTT spamming can cause system reset
- Keypress spamming can cause system reset
- Keypress spamming caused one instance of display overlay issue on secondary head
- Inconsistency in hook switch scanning and hook switch inhibit (hook switch scanning is logically OR'd with inhibit; PTT inhibit is logically ANDed with inhibit)

Day 2

- Brown-out can occur on control heads without any indication to user.
- Ticking sound heard on receiving device while transmitting from another radio (possible test environment issue)
- Ugly degradation in functionality during brown outs
- Power connector can be plugged in backwards
- Whole system impacted when power lost to splitter box
- Two situations where a recoverable brown out was not recovered from.
- After 20 minutes of non-activity system suddenly autoscrolled with no apparently input.

Matt/Raj
Day 1

- Can save a config file without members in the scan group
- Contrast can only be adjusted on one of the heads. (this maybe needs improvement, since it's confusing)
- In emergency mode, plugging a microphone into one of the heads causes mechanical sound to be transmitted

Day 2

- If the volume is down while you scroll the volume is turned up automatically
- The hook switch monitor behavior is not updated when switching networks until hook switch state changes.
- Should not be able to program only one head at once.

Dan/Judy
Day 1

- No way to differentiate which head is which in the selection box or in the device configuration box
- Usage of priority field is unclear and confusing
- The selection box didn't always appear
- Power up seemed too long compared to the single-head config.
- XPA would not read from the lowest priority head
- Didn't reset after programming mode.
- Very inconsistent in reading that an incorrect frequency file was being uploaded to the radio (sometimes error message occurred, and other times not)
- Lots of hanging comms to the radio.
- Name is blank by default
- Primary head inexplicably died, then sprung back into life (after some rattling)
- A channel is not really deleted even after channel delete message is displayed—must back out of the menu
- PTT clunks are audible when audible tones operate on listening head.
- XPA always uses a default setting for the selection box instead of remembering the last setting.

Day 2

- If you've only got one head connected to the XPA and you try to read from the head it will not read.
- Design issue: what is the point of being able to program two heads when there is no scenario where they should be different? Someone should look hard at the database issue. XPA leads you down a path that's WRONG.
- If you are in programming mode and you unplug the secondary head the system does not reset… XPA is in strange state.
- Should function keys stay with the head? Where should this information live?

Raj/Judy
Day 2

- Requirements issue: In emergency mode, if a head drops out, the other head should continue operating.
- During non-stealth emergency mode, if a head drops out, the whole system dies, even if you plug the head back in; if the secondary head is the one that drops out, then the system survives until the end of the current transmit/receive cycle and then dies. (may not occur on first time through… still investigating)
- While in stealth mode, the RSSI level is still being displayed.
- If you define power on state as power on always, then connect it to power, it doesn't power on
- CTCSS tones are audible.

Dan/Matt
Day 2

- Seems strange that there is only one off-hook state for the system.
- When PTT is pressed before removing handset from hook, removal from hook does not put radio in transmit mode until PTT is released and pressed again.

Day 2: Final Session Charter List

Andrew/Josh
- Recon of multi-head functions
- Continuation w/synchronized databases
- Brown-out #1
- Brown-out #2 (this time it's tabular)

Dan/Judy
- Recon of multi-head functions
- Recon of multi-head functions #2
- Explore database capacity and legitimate differences in DB between heads

Matt/Raj
- Recon of multi-head functions
- Recon of multi-head functions #2
- Head disconnect and reconnect

Dan/Matt
- External Speaker functionality

Judy/Raj
- Two-headed emergency mode

Bibliography for Rapid Software Testing

Last revised: November, 2014

Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

Amdahl, Kenn, and Jim Loats. *Calculus for Cats*. Broomfield, Colo.: Clearwater Pub., 2001.

Andrews, Mike, and James A. Whittaker. *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services. Book & CD*. Addison-Wesley Professional, 2006.

Ariely, Dan. *Predictably Irrational, Revised and Expanded Edition: The Hidden Forces That Shape Our Decisions*. 1 Exp Rev. Harper Perennial, 2010.

Austin, Robert D, Tom DeMarco, and Timothy R Lister. *Measuring and Managing Performance in Organizations*. New York: Dorset House Publishing., 1996.

Bach, James. "Testing and Checking Refined." *Testing and Checking Refined*, n.d. http://www.satisfice.com/blog/archives/856.

Bach, James Marcus. *Secrets of a Buccaneer-Scholar: Self-Education and the Pursuit of Passion*. Reprint. Scribner, 2011.

Ball, Philip. *Critical Mass: How One Thing Leads to Another*. 1st ed. Farrar, Straus and Giroux, 2006.

Barabasi, Albert-Laszlo. *Bursts: The Hidden Pattern Behind Everything We Do*. Dutton Adult, 2010.

———. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003.

Baron, Jonathan. *Thinking and Deciding*. 4th ed. Cambridge University Press, 2007.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. 2nd ed. Boston, MA: Addison-Wesley, 2005.

Beizer, Boris. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. 1st ed. Wiley, 1995.

———. *Software Testing Techniques, 2nd Edition*. 2 Sub. Intl Thomson Computer Pr (T), 1990.

Black, David A. *The Well-Grounded Rubyist*. 1st ed. Manning Publications, 2009.

Black, Rex. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. 3rd ed. Wiley, 2009.

———. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. 1st ed. Wiley, 2007.

Bolton, Michael. "Cover or Discover." *Better Software*, November 2008. http://www.developsense.com/articles/2008-11-CoverOrDiscover.pdf.

———. "Got You Covered." *Better Software*, October 2008. http://www.developsense.com/articles/2008-10-GotYouCovered.pdf.

———. "'Merely' Checking or 'Merely' Testing." Accessed September 21, 2012. http://www.developsense.com/blog/2009/11/merely-checking-or-merely-testing/.

———. "Testers: Get Out of the Quality Assurance Business." *Testers: Get Out of the Quality Assurance Business*, n.d. http://www.developsense.com/blog/2010/05/testers-get-out-of-the-quality-assurance-business/.

———. "Three Kinds of Measurement (and Two Ways to Use Them)." *Better Software*, July 2009. http://www.developsense.com/articles/2009-07-ThreeKindsOfMeasurement.pdf.

Boorstin, Daniel J. *The Discoverers*. 1st Vintage Book ed. Vintage, 1985.

Bowman, Sharon L. *Training from the Back of the Room!: 65 Ways to Step Aside and Let Them Learn*. San Francisco, CA: Pfeiffer, 2009.

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Anniversary. Addison-Wesley Professional, 1995.

Brown, John Seely, and Paul Duguid. *The Social Life of Information*. 1st ed. Harvard Business Review Press, 2000.

Burke, James. *Connections*. Simon & Schuster, 2007.

———. *The Pinball Effect: How Renaissance Water Gardens Made the Carburetor Possible - and Other Journeys*. Back Bay Books, 1997.

Carlson, Lucas, and Leonard Richardson. *Ruby Cookbook (Cookbooks*. 1st ed. O'Reilly Media, 2006.

Cayley, David. "How To Think About Science." *How to Think About Science*. Accessed February 23, 2012. http://www.cbc.ca/ideas/episodes/2009/01/02/how-to-think-about-science-part-1---24-listen/.

———. *Ideas on the Nature of Science*. Fredericton, N.B.: Goose Lane Editions, 2009.

Center for History and New Media. "Zotero Quick Start Guide," n.d. http://zotero.org/support/quick_start_guide.

Chabris, Christopher F, and Daniel J Simons. *The Invisible Gorilla: And Other Ways Our Intuitions Deceive Us*. New York: Broadway Paperbacks, 2010.

Chang, Sau Sheong. *Exploring Everyday Things with R and Ruby*. 1st ed. Sebastopol, CA: O'Reilly Media, 2012.

Collins, Harry. *Tacit and Explicit Knowledge*. University Of Chicago Press, 2010.

Collins, Harry M., and Martin Kusch. *The Shape of Actions: What Humans and Machines Can Do*. The MIT Press, 1999.

Collins, H. M. *Rethinking Expertise*. Chicago: University of Chicago Press, 2007.

Collins, H. M, and T. J Pinch. *The Golem : What Everyone Should Know About Science*. Cambridge [England]; New York, NY, USA: Cambridge University Press, 1994.

Cooper, Alan. *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity*. 1st ed. Sams - Pearson Education, 2004.

Cooper, Alan, Robert Reimann, and David Cronin. *About Face 3: The Essentials of Interaction Design*. 3rd ed. Wiley, 2007.

Copeland, Lee. *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

Creswell, John W. *Qualitative Inquiry and Research Design: Choosing Among Five Traditions*. Thousand Oaks: Sage Publications, 2007.

Crispin, Lisa, and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. 1st ed. Addison-Wesley Professional, 2009.

Davis, Wade. *Light at the Edge of the World: A Journey Through the Realm of Vanishing Cultures*. Douglas & McIntyre, 2007.

———. *One River : Explorations and Discoveries in the Amazon Rain Forest*. New York: Touchstone, 1997.

———. *The Wayfinders: Why Ancient Wisdom Matters in the Modern World*. Toronto: House of Anansi Press, 2009. http://site.ebrary.com/id/10488286.

Davis, Wade, K. David Harrison, and Catherine Herbert Howell, eds. *Book of Peoples of the World: A Guide to Cultures*. 2nd ed. National Geographic, 2008.

DeMarco, Tom. "Slack: Getting Past Burnout, Busywork and the Myth of Total Efficiency." Accessed January 27, 2013. http://www.amazon.com/Slack-Getting-Burnout-Busywork-Efficiency/dp/0932633617.

———. *Why Does Software Cost So Much? And Other Puzzles of the Information Age*. New York, N.Y.: Dorset House Pub., 1995.

Dhanjani, Nitesh. *Hacking: The Next Generation*. 1st ed. Beijing ; Sebastopol, CA: O'Reilly, 2009.
Dorner, Dietrich. *The Logic Of Failure: Recognizing And Avoiding Error In Complex Situations*. 1st ed. Basic Books, 1997.
Dyson, George. *Turing's Cathedral : The Origins of the Digital Universe*. New York: Vintage Books, 2012.
Edgren, Rikard. "Questions That Testing Constantly Help Answering | Thoughts from the Test Eye." Accessed February 23, 2012. http://thetesteye.com/blog/2010/01/questions-that-testing-constantly-help-answering/.
Elkins, James. *Why Art Cannot Be Taught: A Handbook for Art Students*. Urbana: University of Illinois Press, 2001.
Engebretson, Pat. *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Syngress the Basics. Waltham, MA: Syngress, 2011.
Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004.
Federman, Mark, and Derrick deKerchkhove. *McLuhan for Managers: New Tools for New Thinking*. Viking Canada, n.d.
Few, Stephen. *Information Dashboard Design: The Effective Visual Communication of Data*. 1st ed. O'Reilly Media, 2006.
Feyerabend, Paul. *Against Method*. Fourth Edition. Verso, 2010.
Feynman, Richard P. *The Pleasure of Finding Things Out: The Best Short Works of Richard P. Feynman*. Edited by Jeffrey Robbins. Basic Books, 2005.
———. *What Do You Care What Other People Think?: Further Adventures of a Curious Character*. Edited by Ralph Leighton. W. W. Norton & Company, 2001.
Feynman, Richard P., and Ralph Leighton. *Surely You're Joking, Mr. Feynman!*. Edited by Edward Hutchings. First THUS Edition. W. W. Norton & Company, 1997.
Fiedler, Rebecca L., and Cem Kaner. "Putting the Context in Context-Driven Testing (an Application of Cultural Historical Activity Theory)," 2009.
Fischer, Edward, Ph.D. *Peoples and Cultures of the World*, Peoples and Cultures of the World. http://www.thegreatcourses.com/tgc/courses/course_detail.aspx?cid=4617.
Fleischman, Michael, and Deb Roy. "Why Verbs Are Harder to Learn than Nouns: Initial Insights from a Computational Model of Intention Recognition in Situated Word Learning." In *Proceedings of the 27th Annual Meeting of the Cognitive Science Society*. Accessed September 27, 2012. http://media.mit.edu/cogmac/publications/cogsci_2005_final.pdf.
Flyvbjerg, Bent. *Making Social Science Matter: Why Social Inquiry Fails and How It Can Succeed Again*. Oxford, UK ; New York: Cambridge University Press, 2001.
Friedl, Jeffrey E.F. *Mastering Regular Expressions*. Third Edition. O'Reilly Media, 2006.
Galison, Peter. *Einstein's Clocks, Poincare's Maps: Empires of Time*. W. W. Norton & Company, 2004.
Gall, John. *The Systems Bible: The Beginner's Guide to Systems Large and Small*. General Systemantics Pr/Liberty, 2003.
Gatto, John Taylor. *Dumbing Us Down: The Hidden Curriculum of Compulsory Schooling*. 2nd ed. New Society Publishers, 2002.
———. *Weapons of Mass Instruction: A Schoolteacher's Journey Through the Dark World of Compulsory Schooling*. Paperback Edition. New Society Publishers, 2010.
Gause, Donald C. *Are Your Lights On?: How to Figure Out What the Problem Really Is*. New York, NY: Dorset House Pub, 1990.

Gause, Donald C., and Gerald Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House, 2011.
Gawande, Atul. *Better: A Surgeon's Notes on Performance*. 1st ed. Picador, 2008.
———. *Complications: A Surgeon's Notes on an Imperfect Science*. Picador, 2003.
———. *The Checklist Manifesto: How to Get Things Right*. First Edition. Picador, 2011.
Gigerenzer, Gerd. *Calculated Risks: How to Know When Numbers Deceive You*. 1st ed. Simon & Schuster, 2003.
———. *Gut Feelings: The Intelligence of the Unconscious*. Reprint. Penguin (Non-Classics), 2008.
Gigerenzer, Gerd, Peter M. Todd, and ABC Research Group. *Simple Heuristics That Make Us Smart*. 1st ed. Oxford University Press, USA, 2000.
Gilbert, Daniel. *Stumbling on Happiness*. Vintage, 2007.
Gladwell, Malcolm. *Blink: The Power of Thinking Without Thinking*. Back Bay Books, 2007.
———. *Outliers: The Story of Success*. Reprint. Back Bay Books, 2011.
———. *The Tipping Point: How Little Things Can Make a Big Difference*. Back Bay Books, 2002.
Gleick, James. *The Information: A History, a Theory, a Flood*. Pantheon, 2011.
Goldratt, Eliyahu M, and Jeff Cox. *The Goal : A Process of Ongoing Improvement*. Great Barrington, MA: North River Press, 2008.
Gonick, Larry. *The Cartoon Guide to Statistics*. 1st HarperPerennial ed. New York, NY: HarperPerennial, 1993.
Gregory, Janet. *More Agile Testing: Learning Journeys for the Whole Team*. Upper Saddle River, NJ: Addison-Wesley, 2015.
Groopman, Jerome. *How Doctors Think*. Reprint. Mariner Books, 2008.
Hallinan, Joseph T. *Why We Make Mistakes: How We Look Without Seeing, Forget Things in Seconds, and Are All Pretty Sure We Are Way Above Average*. Reprint. Broadway, 2010.
Hamilton, James. "Perspectives - Observations on Errors, Corrections, & Trust of Dependent Systems." *Perspectives*, February 26, 2012. http://perspectives.mvdirona.com/2012/02/26/ObservationsOnErrorsCorrectionsTrustOfDependentSystems.aspx.
Harford, Tim. *Adapt: Why Success Always Starts with Failure*. [Toronto]: Anchor Canada, 2012.
Heath, Chip, and Dan Heath. *Made to Stick: Why Some Ideas Survive and Others Die*. 1st ed. Random House, 2007.
———. *Switch: How to Change Things When Change Is Hard*. 1st ed. Crown Business, 2010.
Hendrickson, Elisabeth. *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*. Dallas, Texas: The Pragmatic Bookshelf, 2013.
Higgins, Peter M. *Nets, Puzzles, and Postmen: An Exploration of Mathematical Connections*. Oxford University Press, USA, 2009.
Hofstadter, Douglas R. *Gödel, Escher, Bach: An Eternal Golden Braid*. 20 Anv. Basic Books, 1999.
———. *I Am a Strange Loop*. Reprint. Basic Books, 2008.
———. *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking*. New York: Basic Books, 2013.
"How Software Is Built (Quality Software): Gerald M. Weinberg: Amazon.com: Kindle Store." Accessed August 26, 2012. http://www.amazon.com/How-Software-Built-Quality-ebook/dp/B004KAB9RO/ref=la_B000AP8TZ8_1_54?ie=UTF8&qid=1345979366&sr=1-54.
"How to Observe Software Systems (Quality Software): Gerald M. Weinberg: Amazon.com: Kindle Store." Accessed August 26, 2012. http://www.amazon.com/Observe-Software-Systems-Quality-ebook/dp/B004LDLCAE/ref=la_B000AP8TZ8_1_55?ie=UTF8&qid=1345979617&sr=1-55.

Hubbard, Douglas W. *How to Measure Anything: Finding the Value of Intangibles in Business*. 2nd ed. Wiley, 2010.
Huff, Darrell. *How to Lie with Statistics*. W. W. Norton & Company, 1993.
Hunt, Andrew, and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. 1st ed. Addison-Wesley Professional, 1999.
Illich, Ivan. *Deschooling Society*. Marion Boyars Publishers Ltd, 2000.
Johansson, Frans. *The Click Moment : Seizing Opportunity in an Unpredictable World*. New York: Portfolio/Penguin, 2012.
Johnson, Steven. *The Invention of Air: A Story Of Science, Faith, Revolution, And The Birth Of America*. Reprint. Riverhead Trade, 2009.
Jorgensen, Paul. *Software Testing: A Craftsman's Approach*. Fourth edition. Boca Raton, [Florida]: CRC Press, Taylor & Francis Group, 2014.
Kahneman, Daniel. *Thinking, Fast and Slow*. Penguin, 2011.
Kahneman, Daniel, Paul Slovic, and Amos Tversky, eds. *Judgment under Uncertainty: Heuristics and Biases*. 1st ed. Cambridge University Press, 1982.
Kaner, Cem. *Bad Software: What to Do When Software Fails*. New York: Wiley, 1998.
———. *Foundations of Software Testing*. 1st Ed. Palm Bay, FL: Context-Driven Press, 2014.
———. "Publications « Cem Kaner, J.D., Ph.D." Accessed February 22, 2012. http://kaner.com/?page_id=7.
———. "Software Testing as a Social Science." presented at the STEP 2008, Memphis, May 2008. http://www.kaner.com/pdfs/KanerSocialScienceSTEP.pdf.
Kaner, Cem, and Bach, James. "BBST Foundations." Accessed February 22, 2012. http://www.testingeducation.org/BBST/foundations/.
Kaner, Cem, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. 1st ed. Wiley, 2001.
Kaner, Cem, and Walter P. Bond. "Software Engineering Metrics: What Do They Measure and How Do We Know?," 2004. http://www.kaner.com/pdfs/metrics2004.pdf.
Kaner, Cem, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software, 2nd Edition*. 2nd ed. Wiley, 1999.
Kaner, Cem, Sowmya Padmanabhan, and Doug Hoffman. *The Domain Testing Workbook*. Palm Bay, FL: Context-Driven Press, 2013.
Keirsey, David. *Please Understand Me II: Temperament, Character, Intelligence*. 1st ed. Del Mar, CA: Prometheus Nemesis, 1998.
Kernighan, Brian W., and Rob Pike. *The Practice of Programming*. 1st ed. Addison-Wesley Professional, 1999.
Kernighan, Brian W., and Dennis M. Ritchie. *C Programming Language*. 2nd ed. Prentice Hall, 1988.
Kirk, Jerome, and Marc L Miller. *Reliability and Validity in Qualitative Research*. Beverly Hills: Sage Publications, 1986. http://www.amazon.com/Reliability-Validity-Qualitative-Research-Methods/dp/0803924704.
Klahr, David. *Exploring Science: The Cognition and Development of Discovery Processes*. A Bradford Book, 2002.
"KlaymanHa1987.pdf." Accessed January 27, 2013. http://www.stats.org.uk/statistical-inference/KlaymanHa1987.pdf.
Koen, Billy Vaughn. *Discussion of the Method: Conducting the Engineer's Approach to Problem Solving*. Oxford University Press, USA, 2003.

Krug, Steve. *Don't Make Me Think: A Common Sense Approach to Web Usability, 2nd Edition*. 2nd ed. New Riders Press, 2005.

———. *Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability Problems*. 1st ed. New Riders Press, 2009.

Kuhn, Thomas S. *The Structure of Scientific Revolutions*. 3rd ed. University Of Chicago Press, 1996.

Lehrer, Jonah. "Are Emotions Prophetic? | Wired Science | Wired.com," 01 2012. http://www.wired.com/wiredscience/2012/03/are-emotions-prophetic/.

———. *How We Decide*. 1 Reprint. Mariner Books, 2010.

———. *Proust Was a Neuroscientist*. Reprint. Mariner Books, 2008.

"Less Wrong." Accessed February 18, 2013. http://lesswrong.com/.

Levitin, Daniel J. *The Organized Mind: Thinking Straight in the Age of Information Overload*. New York, N.Y: Dutton, 2014.

Levy, David A. *Tools of Critical Thinking: Metathoughts for Psychology*. 2nd ed. Waveland Pr Inc, 2009.

Lohr, Steve. *Go To*. New York: BasicBooks, 2001.

Marick, Brian. *Everyday Scripting with Ruby: For Teams, Testers, and You*. 1st ed. Pragmatic Bookshelf, 2007.

———. *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*. 1st ed. Prentice Hall, 1994.

Martin, Robert C., ed. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009.

Maxwell, Joseph Alex. *Qualitative Research Design: An Interactive Approach*. Thousand Oaks, Calif.: Sage Publications, 2005.

McLuhan, Marshall, and W. Terrence Gordon. *Understanding Media: The Extensions of Man : Critical Edition*. Critical. Gingko Press, 2003.

Meadows, Donella H. *Thinking in Systems: A Primer*. Chelsea Green Publishing, 2008.

Menand, Louis. *The Metaphysical Club: A Story of Ideas in America*. 1st ed. Farrar, Straus and Giroux, 2002.

Mighton, John. *The End of Ignorance: Multiplying Our Human Potential*. Vintage Canada, 2008.

———. *The Myth of Ability: Nurturing Mathematical Talent in Every Child*. Original. Walker & Company, 2004.

Mlodinow, Leonard. *The Drunkard's Walk: How Randomness Rules Our Lives*. Reprint. Vintage, 2009.

Morville, Peter, and Louis Rosenfeld. *Information Architecture for the World Wide Web: Designing Large-Scale Web Sites*. Third Edition. O'Reilly Media, 2006.

Myers, Glenford J. *The Art of Software Testing*. 1st ed. Wiley, 1979.

Nachmanovitch, Stephen. *Free Play: Improvisation in Life and Art*. New York: G.P. Putnam's Sons, 1990.

Norman, Donald A. *The Design of Everyday Things*. Basic Books, 2002.

Oberg, James. "Did Bad Memory Chips Down Russia's Mars Probe? - IEEE Spectrum." *IEEE Spectrum*, February 16, 2012. http://spectrum.ieee.org/aerospace/space-flight/did-bad-memory-chips-down-russias-mars-probe.

Ogilvy, David. "How to Write." Accessed May 28, 2012. http://www.listsofnote.com/2012/02/how-to-write.html.

Page, Alan, Ken Johnston, and Bj Rollison. *How We Test Software at Microsoft*. 1st ed. Microsoft Press, 2008.

Pascale, Richard, Jerry Sternin, and Monique Sternin. *The Power of Positive Deviance: How Unlikely Innovators Solve the World's Toughest Problems*. Harvard Business Review Press, 2010.

Pellis, Sergio, and Vivien Pellis. *The Playful Brain: Venturing to the Limits of Neuroscience*. Richmond: Oneworld, 2011.

Perry, William E., and Randall W. Rice. *Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*. Dorset House, 1997.

"Personal Names around the World." Accessed January 28, 2013. http://www.w3.org/International/questions/qa-personal-names.

Petroski, Henry. *The Evolution of Useful Things: How Everyday Artifacts-From Forks and Pins to Paper Clips and Zippers-Came to Be as They Are*. 1ST ed. Vintage, 1994.

———. *To Engineer Is Human: The Role of Failure in Successful Design*. Vintage, 1992.

Petzold, Charles. *Code: The Hidden Language of Computer Hardware and Software*. First Paperback Edition. Microsoft Press, 2000.

Pine, Chris. *Learn to Program, Second Edition*. Second Edition. Pragmatic Bookshelf, 2009.

Pink, Daniel H. *A Whole New Mind: Why Right-Brainers Will Rule the Future*. Rep Upd. Riverhead Trade, 2006.

Pinker, Steven. *The Stuff of Thought: Language as a Window into Human Nature*. Reprint. Penguin (Non-Classics), 2008.

Pollan, Michael. *The Omnivore's Dilemma: A Natural History of Four Meals*. Penguin, 2007.

Polya, George. *How To Solve It: A New Aspect of Mathematical Method*. Ishi Press, 2009.

Popper, Karl. *Conjectures and Refutations: The Growth of Scientific Knowledge*. 2nd ed. Routledge, 2002.

Regehr, John. "Embedded in Academia : How to Debug." *How to Debug*, March 1, 2013. http://blog.regehr.org/archives/199.

Reilly, Mary, ed. *Play as Exploratory Learning: Studies of Curiosity Behavior*. Sage Publications, Inc, 1974.

Robinson, Ken, and Lou Aronica. *The Element: How Finding Your Passion Changes Everything*. Reprint. Penguin (Non-Classics), 2009.

"Rule Discovery 22 May 08.pdf." Accessed January 27, 2013. http://forum.johnson.cornell.edu/faculty/russo/Rule%20Discovery%2022%20May%2008.pdf.

Sanitt, Nigel. *Science as a Questioning Process*. Bristol, UK ; Philadelphia: Institute of Physics Pub, 1996.

Schlosser, Eric. *Fast Food Nation: The Dark Side of the All-American Meal*. First Harper Perennial Edition. Harper Perennial, 2005.

Schneier, Bruce. "Schneier on Security: Teaching the Security Mindset." Accessed September 21, 2012. http://www.schneier.com/blog/archives/2012/06/teaching_the_se.html.

Schulz, Kathryn. *Being Wrong : Adventures in the Margin of Error*. New York: Ecco, 2010.

Scott, James C. *Seeing Like a State: How Certain Schemes to Improve the Human Condition Have Failed*. New edition. Yale University Press, 1999.

Senge, Peter M. *The Fifth Discipline: The Art & Practice of The Learning Organization*. Crown Business, 2006. http://www.amazon.com/Fifth-Discipline-Practice-Learning-Organization/dp/0385517254/ref=sr_1_1?s=books&ie=UTF8&qid=1348237089&sr=1-1&keywords=Peter+Senge.

Senge, Peter M., Art Kleiner, Charlotte Roberts, Rick Ross, and Bryan Smith. *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization*. 1st ed. Crown Business, 1994.

Sextus. *Outlines of Scepticism*. Cambridge Texts in the History of Philosophy. Cambridge, U.K. ; New York: Cambridge University Press, 2000.

Shadish, William R., Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. 2nd ed. Wadsworth Publishing, 2001. http://www.amazon.com/Experimental-Quasi-Experimental-Designs-Generalized-Inference/dp/0395615569.

Shapin, Steven, and Simon Schaffer. *Leviathan and the Air-Pump: Hobbes, Boyle, and the Experimental Life*. Reprint. Princeton University Press, 2011.

Silver, Nate. *The Signal and the Noise: Why So Many Predictions Fail--But Some Don't*. New York: Penguin Press, 2012.

Simon, Herbert A. *Administrative Behavior: A Study of Decision-Making Processes in Administrative Organizations*. 4th ed. New York: Free Press, 1997.

———. *The Sciences of the Artificial - 3rd Edition*. Third edition. The MIT Press, 1996.

Stebbins, Robert A. *Exploratory Research in the Social Sciences*. Thousand Oaks, Calif.: Sage Publications, 2001.

Stephens, Matt, and Doug Rosenberg. *Extreme Programming Refactored : The Case Against XP*. Berkeley, Calif.; Berlin: Apress ; Springer, 2003.

Stevenson, John. "Are Testers Ethnographic Researchers?" *The Expected Result Was 42. Now What Was the Test?*, January 18, 2011. http://steveo1967.blogspot.com/2011/01/are-testers-ethnographic-researchers.html.

———. "What You Believe Might Not Be True. (Part 1)." *The Expected Result Was 42. Now What Was the Test?*, January 26, 2011. http://steveo1967.blogspot.com/2011/01/what-you-believe-might-not-be-true-part.html.

Sussman, Noah. "Falsehoods Programmers Believe About Time." Accessed June 19, 2012. http://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time.

Svenson, Ola, and A. John Maule, eds. *Time Pressure and Stress in Human Judgment and Decison Making*. New York: Plenum Press, 1993.

Taleb, Nassim. *Antifragile: Things That Gain from Disorder*. New York: Random House, 2012.

Taleb, Nassim Nicholas. *Fooled by Randomness: The Hidden Role of Chance in Life and in the Markets*. 2 Updated. Random House, 2008.

———. *The Black Swan: Second Edition: The Impact of the Highly Improbable: With a New Section: "On Robustness and Fragility."* 2nd ed. Random House Trade Paperbacks, 2010.

Tavris, Carol, and Elliot Aronson. *Mistakes Were Made (But Not by Me): Why We Justify Foolish Beliefs, Bad Decisions, and Hurtful Acts*. Reprint. Mariner Books, 2008.

Teller. "Teller Reveals His Secrets." *Smithsonian.com*, March 2012. http://www.smithsonianmag.com/arts-culture/Teller-Reveals-His-Secrets.html.

Thomas, Dave, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. 3rd ed. Pragmatic Bookshelf, 2009.

Tittle, Peg. *What If--: Collected Thought Experiments in Philosophy*. New York: Pearson/Longman, 2005.

Tufte, Edward R. *Envisioning Information*. Graphics Pr, 1990.

———. *The Visual Display of Quantitative Information*. 2nd ed. Graphics Pr, 2001.

Watts, Duncan J. *Everything Is Obvious: *Once You Know the Answer*. Crown Business, 2011.

Weick, Karl E. *Sensemaking in Organizations*. Sage Publications, Inc, 1995.
Weinberg, Gerald M. *An Introduction to General Systems Thinking*. 25 Anv. Dorset House, 2001.
———. *Becoming a Technical Leader: An Organic Problem-Solving Approach*. Dorset House Publishing, 1986.
———. *More Secrets of Consulting: The Consultant's Tool Kit*. 1st ed. Dorset House, 2001.
———. *Perfect Software and Other Illusions About Testing*. Dorset House, 2008.
———. *Quality Software Management, Vol. 1: Systems Thinking*. Dorset House, 1991.
———. *Quality Software Management, Vol. 2: First-Order Measurement*. Dorset House, 1993.
———. *Quality Software Management, Vol. 3: Congruent Action*. Dorset House, 1994.
———. *Quality Software Management, Vol. 4: Anticipating Change*. Dorset House, 1997.
———. "Responding to Significant Software Events (Quality Software Management)." Accessed August 26, 2012. http://www.amazon.com/Responding-Significant-Software-Quality-ebook/dp/B004LDM18Q/ref=la_B000AP8TZ8_1_57?ie=UTF8&qid=1345979643&sr=1-57.
———. *The Secrets of Consulting: A Guide to Giving and Getting Advice Successfully*. 1st ed. Dorset House Publishing, 1986.
———. *Weinberg on Writing: The Fieldstone Method*. Dorset House, 2005.
Weinberg, Gerald M., and Daniela Weinberg. *General Principles of Systems Design*. Dorset House, 1988.
Weinberg, Gerald M., and Gerald M. Weinberg. *The Psychology of Computer Programming: Silver Anniversary Edition*. Anl Sub. Dorset House, 1998.
Weiss, John. *The Calculus Direct : An Intuitively Obvious Approach to a Basic Knowledge of the Calculus for the Casual Observer*. [Scotts Valley, CA]: [CreateSpace], 2009.
Wheelan, Charles J. *Naked Statistics: Stripping the Dread from the Data*. First edition., n.d.
Whittaker, James A. *How to Break Software: A Practical Guide to Testing W/CD*. Addison Wesley, 2002.
Whittaker, James A., and Herbert H. Thompson. *How to Break Software Security*. Addison Wesley, 2003.
Yin, Robert K. *Applications of Case Study Research*. 3rd ed. Thousand Oaks, Calif: SAGE, 2012.