# Rapid Software Testing Appendices

# Table of Contents

Except where noted, all material is by James Bach.

## Rapid Testing Methodology



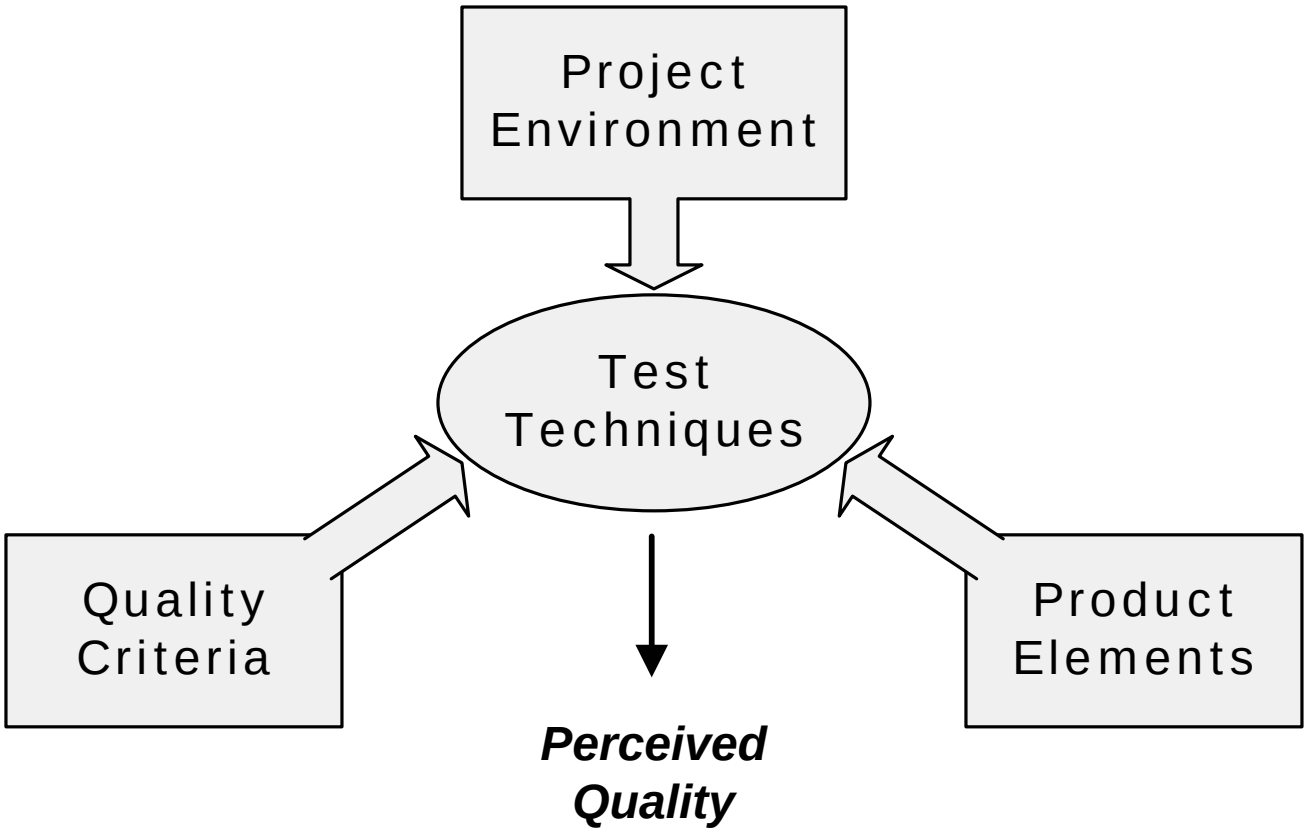
## Rapid Testing Documentation Examples

## Bibliography

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


Version 4.8
3/28/2006

# Heuristic Test Strategy Model




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

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

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

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

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

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

# General Test Techniques

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

## Function Testing

*Test what it can do*

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

## Domain Testing

*Divide and conquer the data*

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

## Stress Testing

*Overwhelm the product*

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

## Flow Testing

*Do one thing after another*

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

## Scenario Testing

*Test to a compelling story*

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

## Claims Testing

*Verify every claim*

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

## User Testing

*Involve the users*

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

## Risk Testing

*Imagine a problem, then look for it.*

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

## Automatic Testing

*Run a million different tests*

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

# Project Environment

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

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

- Do you know who your customers are? Whose opinions matter? Who benefits or suffers from the work you do?
- Do you have contact and communication with your customers? Maybe they can help you test.
- Maybe your customers have strong ideas about what tests you should create and run.
- Maybe they have conflicting expectations. You may have to help identify and resolve those.

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

- Are there any engineering documents available? User manuals? Web-based materials?
- Does this product have a history? Old problems that were fixed or deferred? Pattern of customer complaints?
- Do you need to familiarize yourself with the product more, before you will know how to test it?
- Is your information current? How are you apprised of new or changing information?
- Is there any complex or challenging part of the product about which there seems strangely little information?

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

- *Hubris:* Does the development team seem overconfident about any aspect of the product?
- *Defensiveness:* Is there any part of the product the developers seem strangely opposed to having tested?
- *Rapport:* Have you developed a friendly working relationship with the programmers?
- *Feedback loop:* Can you communicate quickly, on demand, with the programmers?
- *Feedback:* What do the developers think of your test strategy?

## ❑ Test Team. *Anyone who will perform or support testing.*

- Do you know who will be testing?
- Are there people not on the "test team" that might be able to help? People who've tested similar products before and might have advice? Writers? Users? Programmers?
- Do you have enough people with the right skills to fulfill a reasonable test strategy?
- Are there particular test techniques that the team has special skill or motivation to perform?
- Is any training needed? Is any available?

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

- *Hardware:* Do we have all the equipment you need to execute the tests? Is it set up and ready to go?
- *Automation:* Are any test automation tools needed? Are they available?
- *Probes:* Are any tools needed to aid in the observation of the product under test?
- *Matrices & Checklists:* Are any documents needed to track or record the progress of testing?

## ❑ Schedule. *The sequence, duration, and synchronization of project events.*

- *Test Design:* How much time do you have? Are there tests better to create later than sooner?
- *Test Execution:* When will tests be executed? Are some tests executed repeatedly, say, for regression purposes?
- *Development:* When will builds be available for testing, features added, code frozen, etc.?
- *Documentation:* When will the user documentation be available for review?

## ❑ Test Items. *The product to be tested.*

- *Scope:* What parts of the product are and are not within the scope of your testing responsibility?
- *Availability:* Do you have the product to test?
- *Volatility:* Is the product constantly changing? What will be the need for retesting?
- *New Stuff:* What has recently been changed or added in the product?
- *Testability:* Is the product functional and reliable enough that you can effectively test it?
- *Future Releases:* What part of your tests, if any, must be designed to apply to future releases of the product?

## ❑ Deliverables. *The observable products of the test project.*

- *Content:* What sort of reports will you have to make? Will you share your working notes, or just the end results?
- *Purpose:* Are your deliverables provided as part of the product? Does anyone else have to run your tests?
- *Standards:* Is there a particular test documentation standard you're supposed to follow?
- *Media:* How will you record and communicate your reports?

# Product Elements

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

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

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

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

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

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

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

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

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

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

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

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

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

# Quality Criteria Categories

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

## Operational Criteria

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

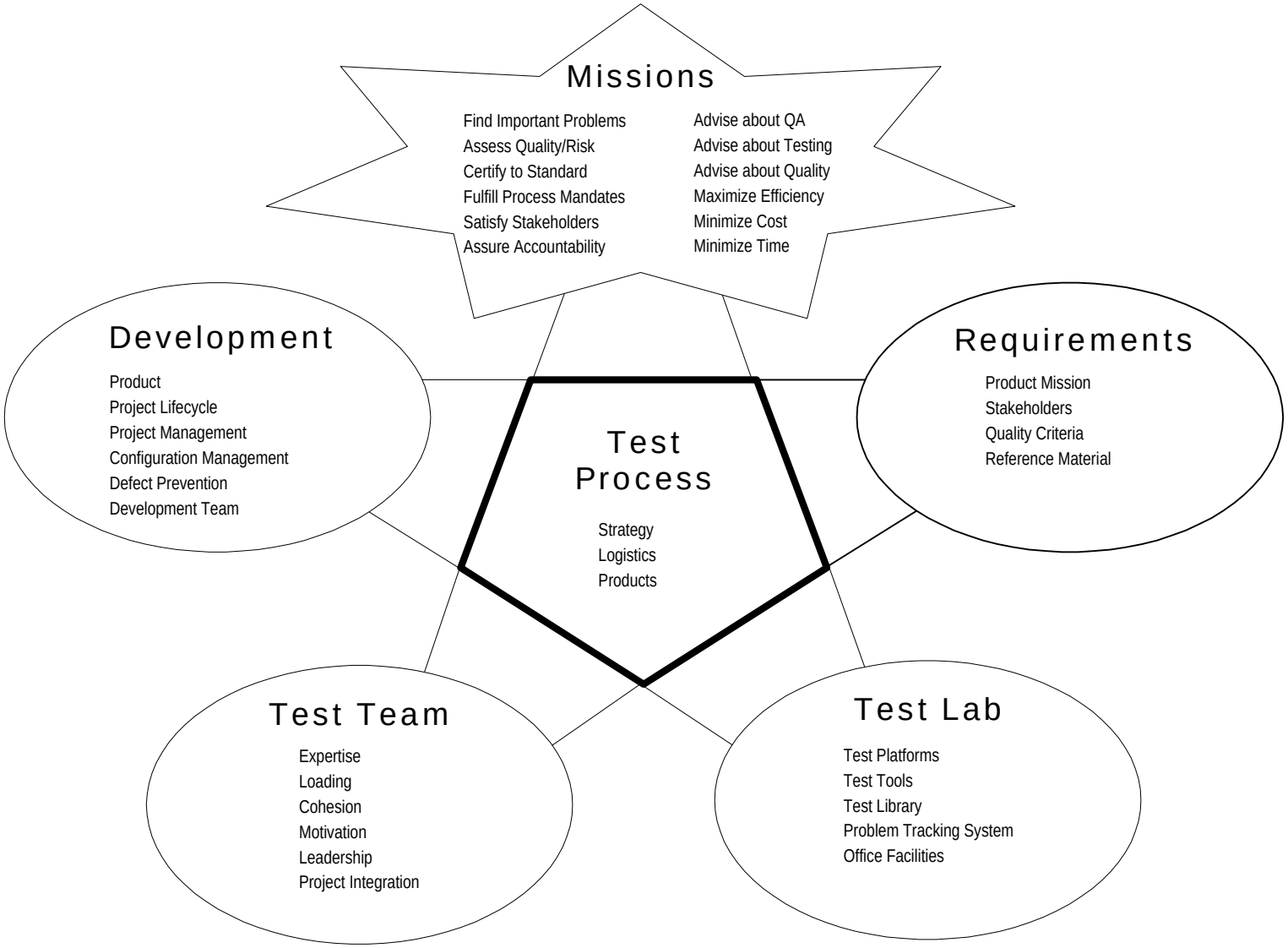
## Development Criteria

- ❑ **Supportability.** *How economical will it be to provide support to users of the product?*
- ❑ **Testability.** *How effectively can the product be tested?*
- ❑ **Maintainability.** *How economical is it to build, fix or enhance the product?*
- ❑ **Portability.** *How economical will it be to port or reuse the technology elsewhere?*
- ❑ **Localizability.** *How economical will it be to adapt the product for other places?*
  - *Regulations*: Are there different regulatory or reporting requirements over state or national borders?
  - *Language:* Can the product adapt easily to longer messages, right-to-left, or ideogrammatic script?
  - *Money:* Must the product be able to support multiple currencies? Currency exchange?
  - *Social or cultural differences:* Might the customer find cultural references confusing or insulting?

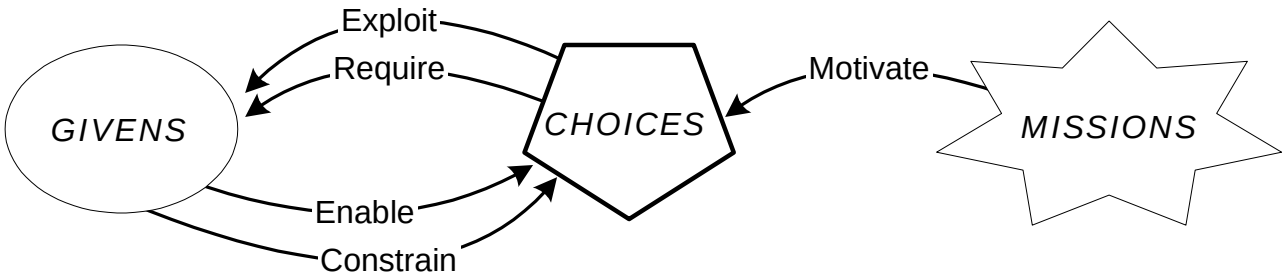
# Heuristic Test Planning: Context Model

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


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 complex or hard to learn?
- ❑ Will testers require special training or tools?
- ❑ Are you remote from the users of the product?
- ❑ Are you remote from any of your clients?
- ❑ Is any part of the test platform difficult to obtain or configure?
- ❑ Will you test unintegrated or semi-operable product components?
- ❑ Are there any particular testability problems?
- ❑ Does the project team lack experience with the product design, technology, or user base?
- ❑ Does testing have to start soon?
- ❑ Is any information needed for planning not yet available?
- ❑ 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?
- ❑ Must you adhere to an unfamiliar test methodology?
- ❑ Are project plans made without regard to testing needs?
- ❑ Is the plan subject to lengthy negotiation or approval?
- ❑ Are project plans changing frequently?
- ❑ Will the plan be subject to audit?
- ❑ Are your clients unsure of what they want from you?

## 2. Clarify 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 to 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. Analyze 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 bemore 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.)
- ❑ Platforms (external hardware and software)
- ❑ Operations (what product's used for)

### Ways to Analyze

- ❑ Perform exploratory testing.
- ❑ Review product and project documentation.
- ❑ Interview designers and users.
- ❑ Compare w/similar products.

### Possible Work Products

- ❑ Test coverage outline
- ❑ Annotated specifications
- ❑ Product 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 found implicit specifications as well as explicit?

## 4. Analyze product 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

- ❑ Review requirements and specifications.
- ❑ Review actual failures.
- ❑ Interview designers and users.
- ❑ Review product against risk heuristics and quality criteria categories.
- ❑ Identify general fault/failure patterns.

### Possible Work Products

- ❑ Component/Risk matrix
- ❑ Risk list

### 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?
- ❑ How will you discover if your risk analysis is accurate?

## 5. Design the test strategy.

*What can you do to test rapidly and effectively based on the best information you have about the product? By all means 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. Let testers use 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. Plan 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 suite management and maintenance
- ❑ Build and transmittal protocol
- ❑ Test cycle administration
- ❑ 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

### Possible Work Products

- ❑ 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?

# General Functionality and Stability Test Procedure

## for Certified for Microsoft Windows Logo

*Desktop Applications Edition*

This document describes the procedure for testing the functionality and stability of a software application (hereafter referred to as "the product") for the purpose of certifying it for Windows 2000. This procedure is one part of the Windows 2000 compatibility certification process described in *Certified for Microsoft Windows Test Plan*.

This procedure employs an exploratory approach to testing, which means that the test cases are not defined in advance, but rather are defined and executed on the fly, while you learn about the product. We chose the exploratory approach because it is the best way to test a product quickly when starting from scratch.

This document consists of five sections:

- **Introduction to Exploratory Testing**
- **Working with Functions**
- **Testing Functionality and Stability**
- **Reading and Using this Procedure**
- **Test Procedure**

The first three parts explain the background and concepts involved in the test procedure. The fourth section gives advice about getting up to speed with the procedure. The fifth section contains the procedure itself.

**This document is designed to be duplex printed (two sides on each page). For that reason, pages 2, 10, and 12 are intentionally blank.**

## Introduction to Exploratory Testing

With this procedure you will walk through the product, find out what it is, and test it. This approach to testing is called *exploratory* because you test while you explore. Exploratory testing is an interactive test process. It is a free-form process in some ways, and has much in common with informal approaches to testing that go by names like ad hoc testing, guerrilla testing, or intuitive testing. However, unlike traditional informal testing, this procedure consists of specific tasks, objectives, and deliverables that make it a systematic process.

In operational terms, exploratory testing is an interactive process of concurrent product exploration, test design, and test execution. The outcome of an exploratory testing session is a set of notes about the product, failures found, and a concise record of how the product was tested. When practiced by trained testers, it yields consistently valuable and auditable results.

The elements of exploratory testing are:

- *Product Exploration.* Discover and record the purposes and functions of the product, types of data processed, and areas of potential instability. Your ability to perform exploration depends upon your general understanding of technology, the information you have about the product and its intended users, and the amount of time you have to do the work.
- *Test Design.* Determine strategies of operating, observing, and evaluating the product.
- *Test Execution.* Operate the product, observe its behavior, and use that information to form hypotheses about how the product works.
- *Heuristics.* Heuristics are guidelines or rules of thumb that help you decide what to do. This procedure employs a number of heuristics that help you decide what should be tested and how to test it.
- *Reviewable Results.* Exploratory testing is a results-oriented process. It is finished once you have produced deliverables that meet the specified requirements. It's especially important for the test results to be reviewable and defensible for certification. As the tester, you must be prepared to explain any aspect of your work to the Test Manager, and show how it meets the requirements documented in the procedure.

## Working with Functions

This procedure is organized around functions. What we call a function is anything the software is supposed to do. This includes anything that results in a display, changes internal or external data, or otherwise affects the environment. Functions often have sub-functions. For instance, in Microsoft Word, the function **`print`** includes the functions **`number of copies`** and **`page range`**.

Since we can't test everything, we must simplify the testing problem by making risk-based decisions about how much attention each function should get. For the purposes of Windows 2000 Certification, you will do this by identifying the functions in the product and dividing them into two categories: *primary* and *contributing*. For the most part, you will document and test primary functions. How functions are partitioned and grouped in the outline is a situational decision. At your discretion (although

the Test Manager makes the ultimate call) a group of contributing functions may be treated as a single primary function, or a single primary function may be divided into primary and contributing sub-functions.

Although you will test all the primary functions, if possible, you may not have enough time to do that. In that case, indicate in your notes which primary functions you tested and which ones you did not test.

It can be hard to identify some functions just by looking at the user interface. Some functions interact directly with the operating system, other programs, or modify files, yet have no effect that is visible on the screen. Be alert for important functions in the product that may be partially hidden.

The functional categories are defined as follows:

| Definition | Notes |
|---|---|
| **Primary Function**<br>Any function so important that, in the estimation of a normal user, its inoperability or impairment would render the product unfit for its purpose. | A function is primary if you can associate it with the purpose of the product *and* it is essential to that purpose.<br><br>Primary functions define the product. For example, the function of adding text to a document in Microsoft Word is certainly so important that the product would be useless without it. Groups of functions, taken together, may constitute a primary function, too. For example, while perhaps no single function on the drawing toolbar of Word would be considered primary, the entire toolbar might be primary. If so, then most of the functions on that toolbar should be operable in order for the product to pass Certification. |
| **Contributing Function**<br>Any function that contributes to the utility of the product, but is not a primary function. | Even though contributing functions are not primary, their inoperability *may* be grounds for refusing to grant Certification. For example, users may be technically able to do useful things with a product, even if it has an "Undo" function that never works, but most users will find that intolerable. Such a failure would violate fundamental expectations about how Windows products should work. |

The first key to determining whether a function is primary is to know the purpose of the product, and that, in turn, requires that you have some sufficiently authoritative source of information from which to deduce or infer that purpose. The second key is knowing that a function is essential. That depends on your knowledge of the normal user, how the function works, and how other functions in the product work.

## Testing Functionality and Stability

Your mission—in other words the reason for doing all this— is to discover if there are any reasons why the product should not be granted Certification, and to observe positive evidence in favor of granting Certification. In order to be Certified for Windows 2000, the product must be basically functional and stable. To evaluate this, you must apply specific criteria of functionality and stability.

These criteria are defined as follows:

| Definition | Pass Criteria | Fail Criteria |
|---|---|---|
| **Functionality**<br>The ability of the product to function. | 1. Each primary function tested is observed to operate in a manner apparently consistent with its purpose, regardless of the correctness of its output. | At least one primary function appears incapable of operating in a manner consistent with its purpose. |
| | 2. Any incorrect behavior observed in the product does not seriously impair it for normal use. | The product is observed to work incorrectly in a manner that seriously impairs it for normal use. |
| **Stability**<br>The ability of the product to continue to function, over time and over its full range of use, without failing or causing failure. | 3. The product is not observed to disrupt Windows. | The product is observed to disrupt Windows. |
| | 4. The product is not observed to hang, crash, or lose data. | The product is observed to hang, crash, or lose data. |
| | 5. No primary function is observed to become inoperable or obstructed in the course of testing. | At least one primary function is observed to become inoperable or obstructed in the course of testing. |

The functionality standard is crafted to be the most demanding standard that can reasonably be verified by independent testers who have no prior familiarity with the product, and only a few days to complete the work. The word "apparently" means "apparent to a tester with ordinary computer skills". As the tester, you will not necessarily be able to tell that the program is functioning "correctly", but if you are able to tell that the product is *not* behaving correctly in a manner that seriously impairs it, the product fails the Certification.

In order to know if the product is seriously impaired for normal use, you must have a notion of what the normal user is like, and what is normal use. In many cases, the normal user can be assumed to be a person with basic computer skills; in other words, someone a lot like the normal tester. In some cases, however, the normal user will be a person with attributes, skills, or expectations that are specialized in some way. You may then have to study the product domain, or consult with the Vendor, in order to make a case that the product should be failed.

In order to perform the stability part of the test, you must also identify and outline the basic kinds of data that can be processed by the product. When testing potential areas of instability, you'll need to use that knowledge to design tests that use challenging input.

## Test Coverage

Test coverage means "what is tested." The following test coverage is required under this procedure:

- *Test all the primary functions that can reasonably be tested in the time available.* Make sure the Test Manager is aware of any primary functions that you don't have the time or the ability to test.
- *Test a sample of interesting contributing functions.* You'll probably touch many contributing functions while exploring and testing primary functions.

- *Test selected areas of potential instability.* As a general rule, choose five to ten areas of the product (an area could be a function or a set of functions) and test with data that seems likely to cause each area to become unstable.

The Test Manager will decide how much time is available for the General Functionality and Stability Test. You have to fit all of your test coverage and reporting into that time slot. As a general rule, you should spend 80% of your time focusing on primary functions, 10% on contributing, and 10% on areas of instability.

Products that interact extensively with the operating system will be tested more intensively than other products. More time will be made available for testing in these cases.

## Sources and Oracles

How do you know what the product is supposed to do? How do you recognize when it isn't working? These are difficult questions to answer outright. But here are two concepts you'll need in order to answer them to the satisfaction of the Test Manager: sources and oracles.

- *Sources.* Sources are where your information comes from. Sources are also what justifies your beliefs about the product. Sometimes your source will be your own intuition or experience. Hopefully, you will have access to at least some product documentation or will have some relevant experience. In some cases, you may need to consult with the Vendor to determine the purposes and functions of the product.
- *Oracles.* An oracle is a strategy for determining whether an observed behavior of the product is or is not correct. An oracle is some device that knows the "right answer." An oracle is the answer to the question "How do you *know* it works?" It takes practice to get good at identifying and reasoning about oracles. The significance of oracles is that they control what kinds of problems you are able to see and report.

Your ability to reason about and report sources and oracles has a lot to do with your qualifications to perform this test procedure. It also helps the Test Manager do his or her job. That's because a poor oracle strategy could cause you to assume that a product works, when in fact it isn't working very well at all. In many cases, you will not have a detailed specification of the product. Even if you had one, you wouldn't have time to read and absorb it all. Still, you and the Test Manager must determine if you can discover enough about the product to access and observe its primary functions. If your sources and oracles aren't good enough, then the Test Manager will have to get the Vendor to assist the test process.

A simple example of an oracle is a principle like this: "12 point print is larger than 8 point print." Or "Text in WordPad is formatted correctly if the text looks the same in Microsoft Word."

One generic pattern for an oracle is what we call the Consistency Heuristics, which are as follows:

- *Consistence with Purpose*: Function behavior is consistent with its apparent purpose.
- *Consistence within Product:* Function behavior is consistent with behavior of comparable functions or functional patterns within the product.
- *Consistence with History:* Present function behavior is consistent with past behavior.

- *Consistence with Comparable Products:* Function behavior is consistent with that of similar functions in comparable products.

Even if you don't have certain knowledge of correct behavior, you may be able to make a case for incorrect behavior based on inconsistencies in the product.

## Reading and Using this Procedure

This procedure follows the pattern of a "forward-backward" process, as opposed to a step-by-step process. What that means is that you will go back and forth among the five different tasks until all of them are complete. Each task influences the others to some degree; thus, each task is more or less concurrent with the others. When all tasks are complete, the whole procedure is complete.

Forward-backward processes are useful in control or search situations. For example, a forward-backward process we're all familiar with is driving a car. When driving, the task of checking the speedometer isn't a sequential step in the process, it's a concurrent task with other tasks such as steering. When driving somewhere, the driver does not just think forward from where he is, but backwards from where he wants to go. Exploratory testing is, in a sense, like driving. Also, like driving, it takes some time, training and practice to develop the skill.

## Task Sheets

This procedure consists of five tasks, which are documented in the Test Procedure section, below. Each task is described by a task sheet with the following elements:

- *Task Description.* Located at the top of each sheet, the task description is a concise description of what you are supposed to do.
- *Heuristics.* In the middle of each sheet is one or more lists of ideas. We call them heuristics. Heuristics are guidelines or rules of thumb that help you decide what to do. They are not sub-tasks that must be "completed." Instead, they are meant to both provoke and focus your thinking. The way to use to them is to visit each idea briefly, and consider its implication for the product you are testing. For example, in the Identify Purposes task, there is a list of potential purpose verbs. One of the ideas on that list is "solve, calculate." When you see that, think about whether one of the purposes of the product is to solve or calculate something. If the product has such a purpose, you might write a purpose statement that includes "Perform various mathematical calculations." If the product has no such purpose, just shrug and move on.
- *Results.* Located at the bottom left of each sheet is a list of what you are expected to deliver as a result of that task.
- *You can say you're done when…* An important issue in a procedure like this is: How do you know when you're done? So, in the bottom right of each task sheet is a list of things that must be true in order for you to be done. In other words, it's not enough simply to produce something that you call a result according to the list at the bottom left. You also have to be prepared to defend the truth of the statements on the right. Most of those statements will require some subjective judgment, but none of them is totally subjective.
- *Frequently Asked Questions.* On the opposite side of each page (this document is designed to be printed two-sided), you'll find a list of answers to questions that testers generally have when first encountering that task.

## The Role of the Test Manager

The Test Manager has ultimate responsibility for the quality of the test process. If any questions are raised about how you tested, the Test Manager must be prepared to vouch for your work. For that reason, escalating issues and questions to the Test Manager is an important part of *your* role.

## Issues and Questions

Issues and questions will pop up during the course of your work. If you can't immediately resolve them without interrupting the flow of your work, then note them and try to resolve them later. These include specific questions, general questions, decisions that must be made, as well any events or situations that have arisen that have adversely impacted your ability to test.

It's important to write down issues and questions you encounter. Your notes may be revisited by another tester, months later, who will be testing the next version of the product. By seeing your issues, that tester may get a better start on the testing. Writing down the issues also gives the Test Manager, or anyone else who reviews your notes, a better ability to understand how the testing was done.

## When to Escalate

In the following situations, ask the Test Manager how to proceed:

- You encounter an obstacle that prevents you from completing one or more of the test tasks.
- You feel lost or confused due to the complexity of the product.
- You feel that you can't learn enough about the product to test it well, within the timeframe you've been given.
- You encounter a problem with the product that appears to violate the functionality or stability standards.
- You feel that the complexity of the product warrants more time for testing than was originally allotted.

## Testing Under Time Pressure

The amount of time allotted to test the product will vary with its complexity, but it will be on the order of hours, not days. Your challenge will be to complete all five tasks in the time allotted. Here are some ideas for meeting that challenge:

- *The first question is whether testing is possible.* Some products are just so complex or unusual that you will not be able to succeed without substantial help from the Vendor. In order to do a good job completing this test procedure on a tight schedule, you first must determine that the job can be done at all.

- *Make a quick pass through all five tasks.* Visit each one and get a sense of where the bulk of the problems and complexities will be. In general, the most challenging part of this process will be identifying and categorizing the product functions.
- *Pause every 20 or 30 minutes*. Assess your progress, organize your notes, and get some of your questions answered.
- *If you feel stuck in one task, try another*. Sometimes working on the second task will help clear up the first one. For instance, walking through the menus of the product often sheds light on the purpose of the product.
- *Tackle hard problems first.* Sometimes clearing up the hard parts makes everything else go faster. Besides, if there's a problem that is going to stop you cold, it's good to find out quickly.
- *Tackle hard problems last.* Alternatively, you could leave some hard problems until later, on the hope that doing an easier task will help you make progress while getting ready to do the rest.
- *Set aside time to clean up your notes.* The final thirty minutes or so of the exploratory test should be set aside for preparing your notes and conclusions for delivery, and doing a final check for any loose ends in your testing.
- *Keep going.* Unless you encounter severe problems or obstacles, keep the process moving. Stay in the flow of it. Write down your questions and issues and deal with them in batches, rather than as each one pops up.

## The Prime Directive: Be Thoughtful and Methodical

Throughout the test procedure, as you complete the tasks, you have lots of freedom about how you do the work. But you must work *methodically*, and follow the procedure. In the course of creating the result for each task, you'll find that you have to make a lot of guesses, and some of them will be wrong. But you must *think*. If you find yourself making wild and uneducated guesses about how the product works, areas of instability, or anything else, stop and talk to the Test Manager.

# Test Procedure

## Complete these five tasks:

- ❑ Identify the purpose of the product.
- ❑ Identify functions.
- ❑ Identify areas of potential instability.
- ❑ Test each function and record problems.
- ❑ Design and record a consistency verification test.

| Things to Deliver | You can say you're done when... |
|---|---|
| ❑ Purpose statement<br>❑ Function outline<br>❑ List of potential instabilities and challenging data<br>❑ Product failures and notes<br>❑ Consistency verification test | ❑ Each task is complete.<br>❑ Each question and issue is either resolved or accepted by the Test Manager.<br>❑ Each task deliverable is accepted by the Test Manager.<br>❑ You know enough about the product to determine whether it should or shouldn't receive Certification according to the functionality and stability criteria. |

# ➢ Identify the purpose of the product.

1. Review the product and determine what fundamental service it's supposed to provide. To the extent feasible, define the audience for the product.

2. Write (or edit) a paragraph that briefly explains the purpose of the product and the intended audience.

---

## Some Potential Purpose Verbs for Use in the Statement

- Create, Edit
- View, Analyze, Report
- Print
- Solve, Calculate
- Manage, Administer, Control
- Communicate, Interoperate
- Serve Data, Provide Access, Search
- Support, Protect, Maintain
- Clean, Fix, Optimize
- Read, Filter, Translate, Convert
- Entertain

## Some Attributes of Users That May be Worth Discussing in the Statement

- Special skills,knowledge, abilities or disabilities
- Troubleshooting ability
- Expectations or needs
- Limitations (who will *not* be a user of this product)

| Things to Deliver | You can say you're done when... |
|---|---|
| ❑ Purpose statement<br>❑ Issues/questions | ❑ You have performed the task as described above.<br>❑ The purpose statement is based on explicit or implicit claims made by the Vendor.<br>❑ All aspects of the product's purpose that are important to a normal user are identified.<br>❑ The purpose statement is fundamental (if it couldn't be fulfilled, the product wouldn't be fit for use). |

# Purposes: Frequently Asked Questions

## Why does this task matter?

Without an understanding of the purposes of the product, you can't defend the distinctions you make between primary and contributing functions. And those distinctions are key, since most of your testing effort will focus on the primary functions. You don't need to write an essay, but you do need to include enough detail so that any function that you think is important enough to call primary can be traced to that statement.

## How do I write a purpose statement?

If the Vendor supplies a product description with the Vendor Questionnaire, start with that and flesh it out as needed. If you have to write it yourself, start with a verb and follow with a noun, as in "edit simple text documents", or "produce legal documents based on input from a user who has no legal training." Also, if there are any special attributes that characterize a normal user of the product, be sure to mention them.

The list of purpose verbs comes from all the purposes gleaned from a review of software on the racks of a large retail software store. It may help you notice purposes of the product that you may otherwise have missed. Similar purpose verbs are grouped together on the list to save space (e.g. calculate, solve), and not because you're supposed to use them together.

## How are purposes different from functions?

Purpose relates to the needs of users. Functions relate to something concrete that is produced or performed by the product.

Sometimes the purpose of a function and the name of the function are the same, as in "print": printing is the purpose of the print function. Most of the time, a function serves a more general goal that you can identify. For instance, the purpose of a word processor is not to search for and replace text; instead search and replace are part of editing a document. Editing is the real purpose. On the other hand, in a product we could imagine called "Super Search and Replace Pro," the search and replace function presumably *is* the purpose of the product.

# ➢ Identify functions.

1. Walk through the product and discover what it does.
2. Make an outline of all primary functions.
3. Record contributing functions that are interesting or borderline primary.
4. Escalate any functions to the Test Manager that you do not know how to categorize, or that you are unable to test.

---

## Some Ways to Look for Functions

- Check online help.
- Check the Vendor Questionnaire.
- Check all programs that comprise the product.
- Check all product menus.
- Check all windows.
- Check toolbars.
- Check all dialog boxes and wizards.
- Right-click on all data objects, interface elements, and window panes (this might reveal context menus).
- Double-click on all data objects, interface elements, and window panes (this might trigger hidden functions).
- Check product options settings for functions that are dormant unless switched on (e.g., automatic grammar checking in Microsoft Word).
- Check for functions that are triggered only by certain input (e.g., saving a JPEG image might trigger a JPEG Save wizard).
- Examine sample data provided with the product.
- Check for error handling and recovery functions that are embedded in other functions.

## Function Classification

- **Primary:** Any function so important that, in the estimation of a normal user, its inoperability or impairment would render the product unfit for its purpose.
- **Contributing:** Any function that contributes to the utility of the product, but is not a primary function.

| Things to Deliver | You can say you're done when... |
|---|---|
| ❑ Function outline<br>❑ Issues/questions | ❑ You have performed enough of the *Identify the Purpose of the Product* task to enable you to correctly categorize functions of the product.<br>❑ You have performed the task as described above.<br>❑ Each primary function you identified is *essential* to the fulfillment of the purpose of the product.<br>❑ You have explored enough to reasonably conclude that all interesting functions of the product are accounted for. |

# Functions: Frequently Asked Questions

## Why does this task matter?

By listing the functions that comprise the operation of the product, you are making an outline of what could be tested. When you complete the testing, this outline is an indicator of what you understood the product to be, and what you might have tested. This outline is an important record for use by the Test Manager or the Vendor as a reference in case they want to question you about what you did and did not do, or by other testers who may test this product in the future.

## What if I'm totally confused as to what are the primary functions?

Escalate to the Test Manager. Do not simply choose arbitrarily. The Test Manager will contact the Vendor for information, locate documentation, or otherwise advise you what to do.

## In what format should I record the functions?

Keep it simple. Use a two- or three-level outline. Record a one-line bullet for each function or functional area. Sometimes a function will not have an official name or label. In that case, make up a name and put it in square brackets to indicate that you invented the name. If there are a hundred functions that all belong one family, list the name of the group as in "Drawing functions," rather than listing each by itself.

If you identify contributing functions, clearly distinguish them from the primary functions.

For example: here is a portion of the function outline for Microsoft Bookshelf:

```
Note...
      Add Current Article
      Delete
      Goto
      Annotation
Search All...
      [Result Outline]
      Articles About...
      Articles Containing the Words...
Find Media...
      All Media...
      Audio...
      Images...
      Animations...
      (Result List)
Go Online...
      BookShelf Premier Search
      BookShelf Premier News
      Encarta Online Library
Advanced Search...
      Books
      Media
      Articles
      [Search Hit Highlighting]
```

# ➢ Identify areas of potential instability.

1. As you explore the product, notice functions that seem more likely than most to violate the stability standards.

2. Select five to ten functions or groups of functions for focused instability testing. You may select contributing functions, if they seem especially likely to fail, but instability in primary functions is more important.

3. Determine what you could do with those functions that would potentially destabilize them. Think of large, complex, or otherwise challenging input.

4. List the areas of instability you selected, along with the kind of data or strategies you'll use to test them.

---

## Some Areas of Potential Instability

- Functions that interoperate with other products (e.g. object linking and embedding, file conversion).
- Functions that handle events external to the application (e.g. wake up a sleeping computer when a fax arrives).
- Functions that make intensive use of memory.
- Functions that interact extensively with the operating system.
- Functions of unusual complexity.
- Functions that change operating parameters (e.g. preference settings)
- Functions that manipulate operating system configuration.
- Functions that intercept or recover from errors.
- Functions that replace basic operating system functions (undelete files or process user logon).
- Any function or set of functions that involve multiple simultaneous processes.
- Functions that manipulate multiple files at once.
- Functions that open files over a network.

## Some Ideas About Challenging Data

- **Documents:** Long documents; a lot of documents open at once; or documents containing lots of different objects.
- **Records:** Long records; large numbers of records, or complex records.
- **Lists:** Long lists; empty lists; multicolumn lists.
- **Fields:** Enter lots of characters; very large values.
- **Objects:** Lots of objects; too many characters; large objects; compound objects.
- **Changes:** Add and delete things; edit without saving or exiting.
- **Loads:** Get a lot of processes going at once; batch processing with large batches; do lots of things in a very short time.
- **Non sequiturs:** Click randomly around windows; type randomly on keys; enter unexpected input.
- **Exceptions and Escapes:** Interrupt processes over and over again; cancel operations; give erroneous data to trigger error handling.

| Things to Deliver | You can say you're done when... |
|---|---|
| ❑ List of potential instabilities and challenging data<br>❑ Issues/questions | ❑ You have completed exploring the product and looking for areas of potential instability.<br>❑ You have performed the task as described above.<br>❑ Everything you identify represents something you will test or have tested.<br>❑ For each potential instability you identify, you can state your reasoning and your sources. |

# Instabilities: Frequently Asked Questions

## Why does this task matter?

When testing for stability, it's a good idea to focus your efforts on areas that are more likely to become unstable. Some input data you give to a product is more likely than others to trigger instability.

## What is instability?

Any behavior that violates the stability standard. Obvious instabilities are crashes. The basic difference between functional failures and instabilities is that, with the latter, the function *can* work but sometimes doesn't. The function is unreliable, but not completely inoperable. It is also often called instability when a function works correctly in some ways, but has negative side effects, such as corrupting some other function or product.

## How do I know what is potentially unstable?

You can't know for sure. The heuristics we provide are general hints. As you explore the product, you may get a feeling about what parts of the product may be unstable. Corroborate your initial suspicions with quick tests. Let's say you suspect that a particular function may harbor instabilities because it's complex and seems to make intensive use of memory. You could corroborate your hypothesis about its complexity just by looking at the complexity of its visible inputs and outputs, and the varieties of its behavior. You could corroborate your hypothesis about memory use by using the Task Manager to watch how that product uses memory as it executes that function.

Once you have a definite idea that a function might be unstable, or at least has attributes that are often associated with instability, design a few tests to overwhelm or "stress" the function. When testing for instability, you don't need to restrict yourself to normal input patterns. However, instabilities exhibited with normal input are certainly very interesting.

# ➢ Test each function and record results.

1. Test all the primary functions you can in the time available.
2. Test all the areas of potential instability you identified.
3. Test a sample of interesting contributing functions.
4. Record any failures you encounter.
5. Record any product notes you encounter. Notes are comments about quirky, annoying, erroneous, or otherwise concerning behavior exhibited by the product that are not failures.

---

## Grounds for Refusing to Certify

- At least one primary function appears incapable of operating in a manner consistent with its purpose.
- The product is observed to work incorrectly in a manner that seriously impairs it for normal use.
- The product is observed to disrupt Windows.
- The product is observed to hang, crash, or lose data.
- A primary function is observed to become inoperable or obstructed in the course of testing.

## Failure Investigation

- Note symptoms of the problem, and justify why it's severe enough to cause failure to Certify.
- Reproduce the problem, if feasible. Provide an estimated percentage of reproducibility.
- Check for additional failures of a similar kind. Determine if this is an isolated case.
- Report to the Test Manager for confirmation.

| Things to Deliver | You can say you're done when... |
|---|---|
| ❑ Product failures<br>❑ Product notes<br>❑ Issues/questions | ❑ You have completed enough of the *Identify Functions* task to know the primary and contributing functions to test, and you have completed enough of the *Identify Areas of Potential Instability* task to know what stability testing to perform.<br>❑ You have performed the task as described above.<br>❑ You have alerted the Test Manager about any primary functions you could not test.<br>❑ You have recorded failures in enough detail to allow the Vendor to reproduce them or otherwise get a clear idea of the symptoms. |

# Test and Record Problems: Frequently Asked Questions

## Why does this task matter?

This is the heart of the whole process. This is the actual testing. The other tasks help you perform this one.

## Wouldn't the process be better if this were the last task to be done?

Only in theory. In practice, testing itself almost always reveals important information about the other tasks that you could not reasonably have discovered any other way. You may think you've completed the other tasks, and then feel the need to revisit them when you're actually testing the functions.

## Why shouldn't I write down all the tests I design and execute?

Although it's a common tenet of good testing to write down tests, the problem is that it takes too much time and interrupts the flow of the testing. If you stop to write down the details of each test, you will end up writing a lot and running very few tests. Besides, it isn't necessary, as long as you can give an overview of what you tested and how, on demand. All the other notes you deliver in this process will help you prepare to do that.

The only test you write down, in this procedure, is the consistency verification test, which represents a small subset of the testing you did.

# ➢ Design and record a consistency verification test.

1. Record a procedure for exercising the most important primary functions of the product to assure that the product behaves consistently on other Windows platforms and configurations.

---

## Consistency Verification Test Requirements

- The test must be specific enough that it can be repeated on the same Windows platform with the same system configuration by different testers, and all testers will get the same results.
- Cover each of the most important primary functions with a simple test.
- Include steps to manipulate graphical objects created within the product.
- Include steps that select objects, drag and drop them.
- Include steps that render and repaint windows across multiple monitors.
- Specify and archive any data needed for the test.
- Specify some complex data for use in the test.
- Use specific file names and path names.
- Make the test as short and simple as you reasonably can, while meeting these requirements.

| Things to Deliver | You can say you're done when... |
|---|---|
| ❑ Consistency verification test<br>❑ Issues/questions | ❑ You have completed enough of the *Identify Functions* task to know which primary functions to include in the consistency verification test.<br>❑ You have performed the task as described above.<br>❑ The test meets all of the requirements listed above.<br>❑ You have executed the test, from beginning to end, exactly as specified. |

# Consistency Verification Test: Frequently Asked Questions

## Why does this task matter?

After the general functionality and stability test is complete, during the rest of the test process there will be an occasional need to perform a simple re-test of functionality and stability. The consistency verification test defines that activity. It's important that this test be precisely defined, because its purpose is to see if changes in Windows platforms or configurations reveal incompatibilities within the product.

## Is this like a "smoke test"?

The term "smoke test" comes from the electronics industry. After a repair, a technician would turn on the device, a television set for example, and look for smoke. The presence of smoke drifting up from the circuit boards told the technician that some parts were getting too much current. If no smoke appeared immediately, the technician would try some simple operations, such as changing the selected channel and volume settings. If the television did those basic functions without any smoke appearing, the technician felt confident to proceed with more specific tests.

The consistency verification test is just like a smoke test, except it's important to define the test with sufficient precision enough that substantially the same test is executed every time.

## How deep should the test be?

Notice what the television technician found out quickly from the smoke test:

- The television set turned on. Picture and sound appeared.
- The basic stuff seemed to work. The user could change channels and turn the volume up and down.
- Nothing burned up.

Notice the detailed tests the technician did not run:

- No attempt to change brightness, contrast or color settings.
- No tests for all possible channels.
- No tests using alternate inputs or outputs.
- No tests using alternate user interfaces (the technician used either controls on the set or the hand-held remote control, but not both).

The consistency verification test you design for the product should verify it at the same level that the technician's smoke tests verified the television set. You should test one example of each major primary function in at least one normal usage.

Another way to think about the test is that it's the set of things you can do with the product that will give the most accurate impression possible of the quality of the product in a few minutes of test time.

# Heuristics of Software Testability

by James Bach, Satisfice, Inc.

## Controllability

***The better we can control it, the more the testing can be automated and optimized.***

- A scriptable interface or test harness is available.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Software modules, objects, or functional layers can be tested independently.

## Observability

***What you see is what can be tested.***

- Past system states and variables are visible or queriable (e.g., transaction logs).
- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected and reported through self-testing mechanisms.

## Availability

***To test it, we have to get at it.***

- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- Product evolves in functional stages (allows simultaneous development and testing).
- Source code is accessible.

## Simplicity

***The simpler it is, the less there is to test.***

- The design is self-consistent.
- Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements)
- Structural simplicity (e.g., modules are cohesive and loosely coupled)
- Code simplicity (e.g. the code is not so convoluded that an outside inspector can't effectively review it)

## Stability

***The fewer the changes, the fewer the disruptions to testing.***

- Changes to the software are infrequent.
- Changes to the software are controlled and communicated.
- Changes to the software do not invalidate automated tests.

## Information

***The more information we have, the smarter we will test.***

- The design is similar to other products we already know.
- The technology on which the product is based is well understood.
- Dependencies between internal, external and shared components are well understood.
- The purpose of the software is well understood.
- The users of the software are well understood.
- The environment in which the software will be used is well understood.
- Technical documentation is accessible, accurate, well organized, specific and detailed.
- Software requirements are well understood.

James Bach, Satisfice, Inc.
v1.24 11/3/2002 11:07 AM
*james@satisfice.com, http://www.satisfice.com*

# 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?**

4.1.1. 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 doc. in any way? screenshots? 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. Was this bug found late in the project? does that indicate a weakness in the test suite?
5.2.2. Will the test automation cover this case?
5.2.3. 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?**

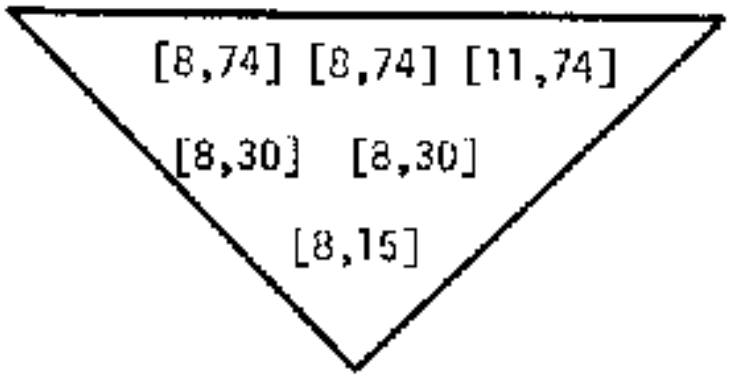
**7.2. What was the error that caused the defect?**

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

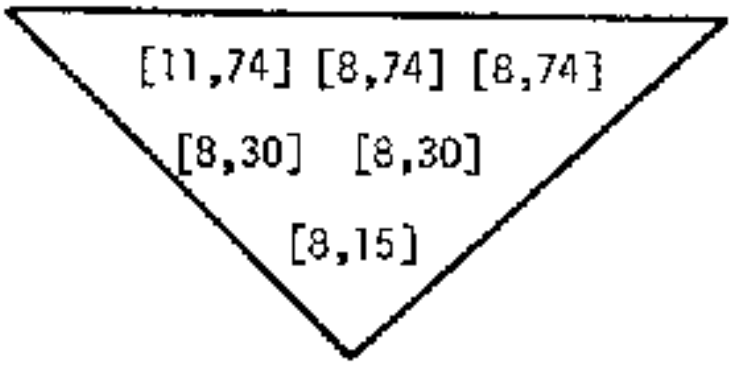
**7.4. Is there any review process that could catch bugs like this before they get into the build?**

---

CDR




LMP




Report Features During Descent And Determine LM Location With HOU (5 Min) Report Angle Of +Z Wrt West. Give General Impression (Earth Analog) And Predominant Features.

SUR-15

Describe Using Monocular: (15 Min)

1 Near Field (define location by angle and distance from LM)

A Features

1 General Surface
2 Plains
3 Craters
4 Rays
5 Cones
6 Boulder Fields
7 Rilles, Faults, Grabens
8 Rock Fragments
9 Loose Ground-Mass Material
10 Coatings

B General Surface

1 Texture - smooth, flat, gentle rolling, rough, jagged

2 Materials - dust, sand, pebbles, rocks, boulders [note size, angularity, and roundness], cinders, ash fall or flow, lava, pahoehoe, aa, ejecta

3 Aerial distribution - uniform, spotted, patterned

4 Color/albedo pattern

5 Contracts - abrupt texture or material changes, color/albedo discontinuities, elevation changes [note sharp or diffuse character]

6 Origin of surface character - cratering, depositional, flow-like

C Plains

1 Extent
2 Degree of cratering (age)
3 Texture - smooth, flat, gentle rolling
4 Color/albedo

D Craters

1 Type - rayed (youngest), blocky rim, sharp rim, low rim, subdued, shallow depressions (oldest), chain, dimple

2 Size/Shape - diameter, depth (dia/depth ratio), circular, polygonal, square, irregular, elongated

3 Ejecta - size, shape, distribution (fields, loops, branches, clusters), material/color/albedo changes, degree of burial

4 Color/albedo pattern

5 Rim - terraced, hummocky, smooth, radial and concentric patterns, flow patterns, boulder or dune fields, small scale color/albedo variations

Basic Date October 27, 1969
Changed

6 Walls - texture, material, small scale color/albedo variations, layers, contacts, strike/dip, bedding, layer thickness and continuity, slump features, flow channels, holes, caves

7 Floor - central peak, eruptive features, radial or concentric flow or fracture patterns, rock/boulder fields, small scale color/albedo variations, spatter

8 Relation to surrounding craters - chain, cluster, random distribution

9 Origin -
Impact: ejecta (direction), central peak, higher rim, rim/wall/floor fragments, impacting material
Volcanic: caldera, flow, cinder, spatter
Collapse: no rim or ejecta evidence of material drainage, similar features along linear faults

E Rays - source, direction, composition, texture/material variations, color/albedo variations, size thickness/width/length ratios

F Boulder Fields - linear, bunched, sloped, size/angularity/roundness/degree of burial

G Rilles, Faults, Grabens

1 Shape - linear, enechelon, angular, sinuous

2 Displacement - relative horizontal and vertical offset of both sides, separation, depth, width

3 Age - angularity and slope of sides, fill at bottom, cratering

4 Color/Albedo variations

5 Walls - texture, material, small scale color/albedo variations, layers, contacts, strike/dip, bedding, layer thickness and continuity, slump features, flow channels, holes/caves

6 Continuity - method of termination, breaks, relative pattern to other similar features, association with other features

H Rock Fragments

1 Size/angularity/roundness

2 Color/albedo relative to surface

3 Height wrt surface - burial, on top, pedestal

4 Surface - visicular, rough, jagged, smooth, layed

5 Distribution - field, cluster, linear group, uniform

I Loose Ground-Mass Material

1 Size - dust, round, gravel, pebbles

2 Sorting - poor, medium, well, bimodal

3 Color/albedo

4 Cohesiveness - loose, friable, cemented, welded

J Coatings

1 Location - windows, LM skin, footpads, rocks, boulders

2 Size - dust, sand, gravel

3 Geometry - uniform, in low spots, rims, fillets, one side only

4 Transport mechanism

# TEST REPORT

## Mission:

1 - Provide information to discriminate between apps. in bake-off WRT who should win.

2 - Help clients develop an understanding of the status of the app., it's strengths and weaknesses, and where it might best be improved.

3 - Develop a productive working relationship with the developer, so he will find it easier to make use of my services and vice versa.

4 - [Personal] Practice agile testing.

## Testing Process:

### DOT NET

- ~~Unsuccessfully tried to log in~~
- Function Testing
- Partial Claims Testing
- Paired Testing w/ opposing developer

### J2EE

- Function testing
- Claims Testing
- Paired testing w/ developer

# Important Testing NOT Done

Need a couple more hours of function & claims testing before diminishing returns point is reached on mission items other than #1

- Client testing not from dev. machine
- Browser compatibility testing
  - Browsers
  - O/S:
  - Browser settings
  - O/S settings
- Stress + Load Testing
  - Performance curve
  - Reliability degradation

Stakeholder conference

Client simulation test framework

# Results

15 minutes

## DOT NET

- No support for Testing

- Benefits
  - Major requirements not implemented or mis-implemented
    - emailing
    - Graphing
    - Technotes
    - Calendar
    - ~~can't submit fault~~
    - ~~Defect list is not filtered by user~~

- Problems
  - Can't Submit Fault
  - Unfiltered defect list
  - Possible refresh problem
  - Can "report" a bug in the future
  - Spelling errors
  - When no technotes, say so.
  - Apparent Sorting feature doesn't work
  - Wrong sized graphic
  - Middle justified

30 minutes

## J2EE

- Anticipated test process
  - Unit test framework
  - Testability interface
  - Met tester and assured readiness of test platform

- Benefits
  - Two requirements not implemented
    - email sending
    - Technotes

Problems

- Developer reports Stuck severity
- middle justification on forms

# Recommendation

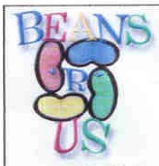
- J2EE app. is substantially more functional in terms of stated and negotiated requirements.
- With the exception of actual email sending and Tech note viewing, the app. appears to have fulfilled each req., although the UI. is as yet rudimentary and, in my opinion, not ready to deploy for real users

# Bake-Off

"Beans-R-Us"
Web-Based Fault Reporting

Your mission, should you choose to accept it, is to develop a web-based course registration system for Beans-R-Us, along with the requisite back-end or in-office support functionality.

Warning: this message will not self-destruct.




## Beans-R-Us

- BroadBean – General ledger
- RunnerBean – Creditor's ledger
- KidneyBean – Debtor's ledger
- SoyaBean – Fixed asset's register
- StringBean – Cash book
- LimaBean – Payroll
- CoffeeBean – Timecard management
- BlackBean - Reporting

Beans-R-Us is a company that sells and maintains a financial package, containing a number of modules within it.


## Components

- Web-based
  - View tech notes
  - Log-in
  - Report fault in a product
  - View status of reported faults
- Office-based
  - E-mail fault details to admin
  - Assign fault to employee
  - Update fault
  - Create outstanding faults summary

### Web-based Functionality

The user must be able to:

- View any tech notes for a particular product.
- Log-in to report a fault or view the status of their reported faults.
- Report a fault in a particular product.
- View the status of any faults that they have reported.

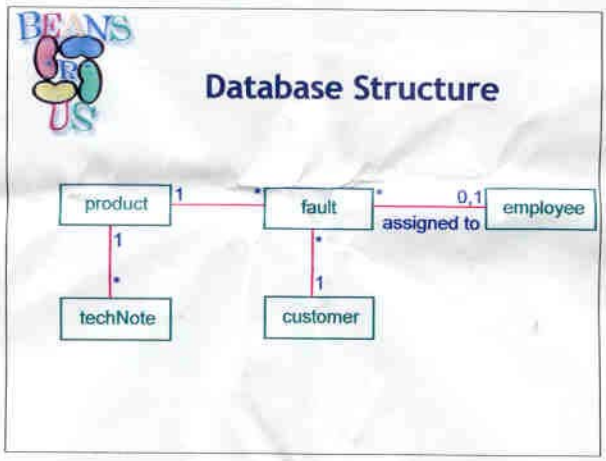
### Office-based Functionality

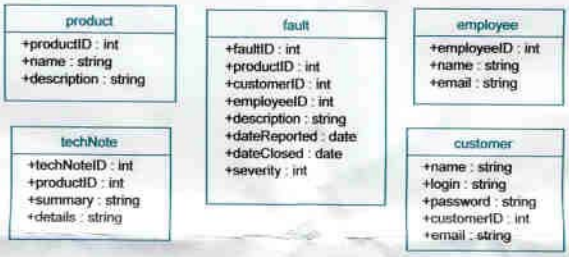
Staff must be able to:

- Assign a fault to an employee, e-mailing the employee about the fault now assigned to him/her.
- Update the details of a fault.
- Create a summary report listing all faults.
- Create a summary report graph showing the numbers of outstanding faults, categorised by how long the fault has been open (less than a week, 1 to 2 weeks, 2 to 4 weeks, longer than 4 weeks).

The system must automatically:

- E-mail the details of any new faults reported to admin@beansrus.com who will assign the fault to a particular employee.


BEANS R US
Database Structure
product
1
*
fault
*
0,1
employee
assigned to
1
*
*
1
techNote
customer


product
+productID : int
+name : string
+description : string
techNote
+techNoteID : int
+productID : int
+summary : string
+details : string
fault
+faultID : int
+productID : int
+customerID : int
+employeeID : int
+description : string
+dateReported : date
+dateClosed : date
+severity : int
employee
+employeeID : int
+name : string
+email : string
customer
+name : string
+login : string
+password : string
+customerID : int
+email : string

4

# Putt Putt Saves the Zoo

**(Test 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???

# 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*

# The Risk of Incompatibility

Software written by one developer or development team often doesn't work with that written by other developers and teams. This problem occurs at all levels of software systems from individual modules to large interoperating systems.

The way we catch integration bugs is through system-level testing, to assure that all parts of a system work together to meet requirements. Thorough system testing is difficult and laborious. Fortunately, the most important compatibility problems reveal themselves quickly, so the process is amenable to a risk-based approach.

## *Compatibility Problems*

| | |
|---|---|
| **Interoperability** | • Version incompatibility of shared DLL's<br>• Incorrect use of sub-system API's<br>• Functional interference between sub-systems<br>• Sub-systems can't share data<br>• One sub-system fails so as to corrupt another sub-system<br>• Sub-system functions are not synchronized |
| **Resource Management** | • O/S resource overload<br>• Poor memory management<br>• Poor storage management |
| **Platform Configuration** | • Sub-systems require mutually exclusive configurations |
| **Usability** | • Inconsistent look and feel among sub-systems<br>• Inconsistent behavior among sub-systems<br>• Poor overall performance<br>• Documentation is disjointed |
| **Installability** | • Installing one sub-system clobbers another sub-system<br>• Installation process is too confusing for typical user<br>• Installation process is disjointed<br>• Upgrading one sub-system causes entire system to fail |
| **Support** | • Customer service personnel are not expert in each sub-system<br>• Users must go to several different sources to get support |

## *Compatibility Risk Factors*

| | |
|---|---|
| **Complexity** | • Of each sub-system<br>• Of entire system |
| **Volatility** | • In interfaces<br>• In design<br>• In implementation |
| **Disorganization** | • Among project schedules<br>• Among project teams<br>• Among companies |
| **Ignorance** | • Confidentiality barriers<br>• Incomplete specifications<br>• Ambiguous specifications<br>• Outdated specifications |
| **Platform Variety** | • Diverse operational profiles<br>• Diverse platform configurations |

# 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.<br>• Review makefiles for simplicity, documentation, and consistency. |
| **Performance** | • Benchmark performance of low level encapsulation and high-order functionality versus:<br>- OWL 1.0x<br>- MFC<br>- Native Windows apps<br>• Actively solicit Beta tester feedback, design questionnaire, tabulate/analyze results. |
| **Internationalization** | • Verify international enabling of the following:<br>- Stored strings (window titles, diagnostics, etc.)<br>- Menus items and accelerators<br>- Cutting and pasting text (clipboard support)<br>- Printing<br>- Localized versions of common dialogs<br>- Status line code<br>- Input validation (proper uppercasing, etc.)<br>- filenames/streaming |
| **Design Quality** | • Inspect code for appropriate use of C++ idioms.<br>• Participate in discussions to promote:<br>- Design simplicity<br>- Backward compatibility<br>- Appropriate feature set<br>- Flexibility for future technologies |
| **Documentation Quality**<br>*Reference Guide* | • Confirm API coverage with latest available header files.<br>• Check completeness of information for each API, member function, and data item.<br>• Review material for overall usability/organization. |
| *Programmer's Guide* | • Check for missing pieces:<br>- Versus MFC<br>- Versus Petzold (native Windows)<br>- Versus our provided examples<br>- Revealed by beta survey feedback<br>- RTL/Classlib functionality used by Elvis<br>- C SDK methods compared with Elvis methods<br>• Review example code versus:<br>- Code style/readability/comprehensibility<br>- Compile-time errors/warnings<br>- Run-time bugs<br>• Review material for overall usability/organization. |

| | |
|---|---|
| *Tutorial* | • Actively solicit feedback from neophyte Elvis users.<br>• Review example code versus:<br>- Code style/readability/comprehensibility.<br>- Compile-time errors/warnings.<br>- Run-time bugs. |
| **Application size and efficiency** | • Benchmark Elvis size (DGROUP, .EXE) and performance vs.:<br>- Elvis 1.0x<br>- MFC<br>- Native Windows apps<br>• Check diagnostics:<br>- Measure effect of varying levels of diagnostics<br>- Determine optimum/shipping versions of final vs. 'debug' libraries, re: size/efficiency<br>• Actively solicit Beta feedback from:<br>- Power Users (substantial/industrial strength apps.)<br>- 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<br>• Verify debugger support for:<br>- Special Elvis needs: entry point/Winmain issues, Elvis diagnostics, etc.<br>- Any debugging problems highlighted by Elvis: heavily templatized code, exceptions, RTTI, linker capacity, etc.<br>• 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:<br>- `#ifdef` or remove Win16-specific calls<br>- `#ifdef` full Win32-specific calls<br>- `#ifdef` Win16 calls which have better Win32/s equivalents<br>• Execute test suites to verify that examples and other suites produce the same output for both static and dynamic libs.<br>• Investigate the following C++ Compilers for Elvis compatibility:<br>- Symantec<br>- MetaWare<br>- Microsoft<br>- CFront<br>• Execute test suites to verify that examples and other suites produce appropriate output for the following (using debug kernel):<br>- Win 3.1<br>- Win32s on Win 3.1<br>- Win32/s on Windows NT<br>- Win 3.1 on Windows NT<br>- Win 3.1 on OS/2<br>• Investigate Elvis compatibility using Mirrors on OS/2. |

| | |
|---|---|
| **High-order functionality**<br>*System level* | • Review specifications to assure that the following functionality is supported:<br>- OLE<br>- VBX<br>- GDI<br>- BWCC<br>- CTRL3D<br>• Track support issues for 3rd party:<br>- Frameworks<br>- Class libraries (Rogue Wave, etc.)<br>- Custom control (widget) collections<br>• Track interoperability issues for Borland products:<br>- Class libraries (Classlib, RTL iostreams, etc.)<br>- Engines (Pdox, BOLE2, etc.)<br>- 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:<br>- Event response tables to replace DDVTs<br>- Windows' resources from multiple DLLs; TLibManager<br>- Document View model<br>- OLE DocFile support<br>- Common dialogs<br>- Clipboard support<br>- Floating palette<br>- Window decorations/gadgets (tool bars/status bars)<br>- Input validation support<br>- Printer support<br>- Use of C++ exceptions<br>- Menus (including OLE 2.0 support)<br>- GDI (fonts, brushes, pens, palettes, bitmaps, regions, icons, cursors, DIBs, complete device context encaps.)<br>- Virtual listboxes (1,000,000,000 items)<br>- Edit control without limits<br>- Outliner/Tree structure listbox<br>- Edit control that will take multiple fonts<br>- Print Preview<br>- Edit control like QPW's<br>- Gauges, sliders, spin buttons, split panes<br>- Example(s) showing use of ODAxxxxx (OwnerDrawAccess APIs)<br>- Workshop aware custom controls (there's already a hack on CIS)<br>- OWL custom control(s) that are usable by 'C SDK' style applications |

| | |
|---|---|
| **Low-level API encapsulation** | • Review message response macros for coverage.<br>• Verify that all appropriate APIs (i.e., OS features) are encapsulated.<br>• Compare item-by-item to MFC and other competitors.<br>• Verify that API functionality is fully accessible and fully usable.<br>• Check internal data structures for completeness.<br>• Verify consistency of Elvis abstractions (i.e., compared to the native API parameter order, data types, etc.).<br>• 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<br>• Assure that OWL 1 apps are upgradeable to Elvis vis-a-vis:<br>- Documentation (usability testing, beta banging, careful in-house review)<br>- Automated conversion tool works intuitively<br>- Usability and documentation of design changes<br>- 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.<br>• Create or collect special test code, including at least one large-scale omnibus application.<br>• Create and maintain smoke tests runnable by Integration.<br>• Build OWL library, after each delivery that has changes in source or include files, for:†<br>- 16bit small static<br>- 16bit medium static<br>- 16bit large static<br>- 16bit large DLL<br>- 32bit flat static<br>- 32bit flat DLL<br>- All of the above in diagnostic/debugging mode.<br>• Build selected models with -Vf, -O2, -xd, -3, -dc and -po:‡<br>- 16bit large/medium static (switch every other time between medium and large)<br>- 16bit large DLL<br>- 32bit flat fully optimized for speed and/or size (if not already delivered that way)<br>• Verify that user built libs are identical to 'delivered' libs (except paths and time stamps).<br>• Build all examples in all models listed above and run automated regressions.<br>• 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

**MEETING NOTES** **APRIL 26, 2000**

# Deployment Planning and Risk Analysis

## How do we deploy Waterfall II safely?

These are the notes from a meeting held from 4-8pm on April 26th. The primary purposes of this meeting were to examine the risks of deploying Waterfall II and to come to a consensus on an overall approach to deployment that would best manage those risks. A secondary purpose of the meeting was to examine the general risks and dynamics of deployment as part of an ongoing effort to improve our ability to deploy future products successfully.

These notes are an edited version of what appeared on the whiteboards and flipcharts during the meeting. The raw notes, exactly as transcribed, are attached.

### Attendees

- Sharon xxx (facilitator and recorder)
- A.C. xxx
- Will xxx
- Craig xxx
- Neal xxx
- Melora xxx
- Satish xxx
- Lisa xxx
- Dave xxx
- James Bach

### Contents

## Meeting Overview

The major events of the meeting followed the spirit of the original agenda. Many issues were raised and concerns discussed, while Sharon kept us focused on the objectives of the meeting. We accomplished the stated mission of the meeting in the four allotted hours. Specifically:

1) Sharon began by clarifying the purposes of the meeting and presenting an agenda.

2) We brainstormed a list of risk drivers, which are problems and challenges that contribute to deployment risk.

3) We brainstormed a list of "nightmare scenarios", which are serious problems or patterns of problems that could befall us during and after an attempt to deploy a new version of the product.

4) As a way of preparing to consider alternative deployment plans, we brainstormed list of risk mitigation ideas for two of the nightmare scenarios. We noticed that many of those ideas would apply to the other scenarios, as well.

5) We brainstormed and discussed a list of alternative deployment strategies. It quickly became apparent that two most viable choices are staged deployment and full deployment.

6) We examined the benefits, risks, and implementation issues associated with each deployment strategy.

7) We came to a consensus that the full deployment strategy involves significantly less risk to execute than the staged deployment option. We also see how we can execute a full deployment with less risk than we've experienced in past deployments. Among the improvements we intend to implement is a more detailed and reliable rollback plan.

8) We identified next steps and assigned action items to the team.

## Risk Drivers of Deployment

The following factors came from our brainstorm. We did not discuss them in much detail, but no driver appears on this list unless it received general assent from the team. The items have been edited into sentence form and reordered by affinity.

- Many complicated changes have been made to the product.
- We have no rollout plan to explain changes to our customers.
- We have no coordinated roll-forward plan for deployment.
- Only one customer is in beta as of 4/26.
- External beta is too brief and has too few customers.

- There is no rollback plan.
- The rollback plan is untested.
- We don't know our criteria for deciding to rollback.
- The baselining process is not stable, and it doesn't meet security requirements.
- We have many more customers, now, who could be impacted.
- The product is used in more and diverse ways than it used to be.
- The new system has an unknown impact on workarounds currently used by customers.
- Our requirements and design may be inadequate. (We may not have correctly understood user needs and usage patterns).
- We have little experience with TRX.
- We have little experience with ClearCase and ClearQuest.
- We have no baseline of information about the performance of the current product.
- We don’t know the performance and reliability of the new system, under load.
- Our performance goals are not specified.
- Our tools for monitoring the reliability of the production system are inadequate.
- Our tools for determining usage patterns are inadequate.
- We have not done a security review of the system.
- Lab machines are not secure.
- We’re relying on third-party testing for some important tests.
- People may get burned out.
- Employee turnover may impact our ability to execute the deployment.
- We don’t have enough hardware to do a staged release.
- Critical hardware may fail.
- There may be unidentified or unmanaged single points of failure in the system.
- The data migration process is not optimized.
- Our database deployment process is not documented or automated.
- Our application and web deployment process is immature.
- Training for Customer Value is minimal.
- Training for maintenance staff is minimal.
- Dependencies with other projects could interfere with deployment.
- Critical maintenance items may interfere with deployment.
- The Firefly code freeze.
- Continual crunch mode could make us complacent about risks.
- Deferred items are sometimes forgotten (the “black hole”).
- Code reviews started late.
- We have no formal freeze process for requirements.
- Important stakeholders (such as QA) are sometimes left out of requirements process.
- Overall, there is little documentation of key processes.
- We have no failover for beta.
- Business requirements seem to change frequently.

## Nightmare Scenarios with Mitigation Ideas

At the beginning of the meeting, it was suggested that deployment risks fell into three categories:

- Time to Market
- Customer Satisfaction
- Technical Integrity

Later, we brainstormed some "nightmare scenarios" that expanded upon those three basic deployment risks. (These have been edited for clarity and redundancy. See the appendix for the raw list):

- We can't get through the deployment in the time required.
- There are a large number of different post-deployment critical escalations.
- There is one critical post-deployment problem so serious that it cripples the system.
- There are problems that make the system seem unusable to most customers.
- There is a data corruption problem or some other problem that is not discovered until it does a lot of damage.
- We lose customers' data.
- The system can't handle the user load.
- TRX does something that corrupts the service.
- We experience a security breach.
- We decide to rollback based on a misunderstanding of the problems in the system.
- Our rollback process fails: it takes too long or it loses customer data.
- It takes us too long to fix critical problems.

For two of these scenarios, we listed some ideas for mitigating the risks:

### We can't get through the deployment in the time required.

- Create detailed deployment plan.
- Practice the plan.
- Estimate the deployment time as accurately as possible.
- Use additional machines so we can abort the deployment and bring old machines back online in case deployment is blocked.
- Use site unavailable screen.
- Insert checkpoints with entry and exit criteria into the deployment plan.
- Assure 7x24 staff availability.

### There are a large number of different post-deployment critical escalations.

- Determine criteria for determining a critical issue.
- Plan in advance what actions we may take and how decisions will be made:
  *how to have a smooth running release status meeting*
  *who can call for a rollback*
  *whose makes the final decision to rollback*
  *who should be in the release status meeting*
- Have resources available to fix issues.

- Have resources available starting at 6am.
- Hold periodic status meetings to assess situation w/IT&CV&ENG&QA.
- Coordinate release w/CV to deploy at non-peak period.

## Deployment Alternatives

We discussed several deployment alternatives and their variations. Eventually, we decided that all of the options were just variations of full deployment and staged deployment:

### Option A: Deploy all at once to production

#### Benefits

- Minimizes time to market.
- No need to maintain parallel code branches.
- Avoids technology enhancements that would be required for staged release.
- Support, development, and testing focus on only one product at a time.
- Reduced operational effort and cost compared to staged release.
- It's the simpler of the two plans—fewer variables to manage.

#### Problems

- We could lose all our customers if there's a big blowup.
- Load hits the new system all at once.

#### Implementation

- Improve the detailed deployment plan.
- Practice the deployment plan.
- Develop a detailed rollback plan.
- Practice the rollback plan.
- Conduct load testing and define acceptance criteria
- Have at least one external beta customer. Three would be better.
- Investigate what would be involved in giving special attention to selected customers during the transition.
- Review the deployment and rollback plans (use notes from 4/26 risk meeting in that review).

### Option B: Deploy in stages to subsets of customers

#### Benefits

- Minimizes impact to customer base in case of trouble.
- Allows management of load ramp up.

### Problems

- Requires concurrent testing and maintenance of old system and new system.
- Potential for longer response time to problems.
- Requires more HW.
- Double deployments.
- We'd have to create a second URL to handle parallel systems, and deal with all the problems associated with that.
- QE98 and ECadmin connectivity would have to be figured out.
- We don't have enough people, presently, to handle all the work.
- Would significantly delay other projects.
- The rollback plan would be more complex.
- We may lose customers because of delay in full rollout.
- We aren't fully meeting customer needs under current operational conditions, let alone the conditions of a staged release.

### Implementation

- We'd need to update Onyx to capture which released system each user is on.
- 30 days to exercise system and meet criteria. (what criteria?)
- We'd deploy first to about 10% of the customer base, then to everybody.
- We'd focus first on "small guys who have a lot of credit card activity"
- We'd deploy in mid-month
- If we deploy new system only to new customers, then the first stage will need to be longer.

## Next Steps

### General

- Find out how other companies manage deployment risks.
- Inform test outsource firm that performance tests are no longer optional to complete by GA.

### Deployment Plan

*Owner:*
- A.C. (will know schedule by end of day Thu)

*Helpers:*
- Satish
- Melora
- Lisa
- Neal

## Rollback Plan

*Owner:*

- A.C. (will start Thu)

*Helpers:*

- Rod
- Rob
- Jill
- Steve

## Load Test

- Create Use Cases
- Create Scripts (outsourced)
- Review load test results for deployment criteria

## Appendix: Raw Notes

```
Deliverable
---------------
- Deployment approach for W2
- Globally... steps for review for major releases

Started by brainstorming risks:

Time to Market
Customer Satisfaction
Technical Integrity


risk drivers:
only 1 customer in beta as of 4/26
lots of complicated changes
untested rollback plan
no rollback plan
unknown or short external beta
not fully stable baseline process
   - doesn't meet security requirements
lots more customers that could be impacted
more diverse usage patterns
don't know our rollback criteria
little experience TRX
no rollout plan to customers-communication
no coordinated roll-forward plan for production deployment
lack of performance information baseline
lack of knowledge about performance/reliability under load
use of 3rd party testing
employee turnover
inadequate reliability monitoring tools
lack of security review
insufficient hardware for staging
no optimized data migration process
no documented /automated database deployment process
immature application/web deployment process
minimum CV training
minimal maintenance training
little documentation
dependencies with other projects
resource burnout
Lack of experience with ClearCase/ClearQuest
critical maintenance items/911
Firefly code freeze
risk complacency
deferred items forgotten
unmitigated single points of failure
lack of adequate requirements & design
potential hardware failure
no formal freeze on requirements
important stakeholders left out of process
lab machines are exposed
code reviews started late
inadequate tools to determine usage patterns
unknown impact on current workarounds
unspecified performance goals
no beta failover
frequently changing business requirements
```

Can't get through the deployment in the time required
Large # of critical escalations
One huge critical issue
something happens that makes the system unusable
delayed discovery of critical problems
Very difficult or time consuming to fix critical problems
TRX does something that corrupts the service
False alarm triggering rollback
data loss
Rollback failure (takes too long, data loss)
security breach
failure of system to handle load

Can't get through the deployment in the time required
- create detailed plan
- practice the plan
- estimate deployment time accurately
- use additional machines so we can sotp and bring up old machines in case deployment is blocked.
- utilize site unavailable screen
- the plan should have checkpoints with entry and exit criteria.
- 7/24 resource availability

Large # of critical escalations
- Criteria for determining a critical issue
- plan action to take when decision point is reached.
  - how to have a smooth running meeting
  - who can call for a rollback
  - whose decision is it to go/no go?
  - who should be in the meeting?
- Have resources available to fix issues
- Have resources available for 6am
- Periodic status meetings to assess situation w/IT&CV&ENG&QA
- calibrate release w/CV to deploy at non-peak period

mitigation master strategies
- deploy everything to production w/rollback plan w/data loss
- deploy everything to production w/rollback plan w/o data loss

  Benefits
  - time to market
  - minimize double maintenance
  - no new product enhancements
  - focused support, development, testing
  - reduced operational efforts/costs

  Issues
  - could lose all customers if there's a big blowup

  Implementation
  - Improve detailed deployment plan
  - practice deployment plan
  - develop detailed rollback plan
  - practice rollback plan
  - conduct load testing & define acceptance criteria
  - have at least one external beta customer/goal of 3
  - investigate what would be involved in handling selected customers during the transition
  - review plans (use the meetings notes from 4/26 risk meeting to help with this)

"An outage is much less of a problem than corrupted data"

- staged deployment to subset of customers
   - what is the duration? What criteria?
   - if only for new customers, then it needs to be longer

   Benefits
   - minimize to customer base
   - allows management of load ramp up

   Implementation
   - 30 days to exercise system and meet criteria
   - est. 10% customer base, then deploy to everybody
   - "small guys who have a lot of credit card activity"
   - deploy in mid-month

   Issues
   - double testing
   - double maintenance
   - potential longer time to respond to issues
   - update Onyx to capture which system users are on
   - more HW required
   - double deployments
   - URL issue-- how to handle
   - don't have enough people, presently
   - delays other projects significantly
   - QE98 and ECadmin connectivity
   - more complex rollback
   - may lose customers because of delay in rollout
   - can't meet customer needs under current conditions, let alone under conditions of a staged release

mitigation variables
- extend beta w/more customers
- comprehensiveness of the rollback plan

NEXT STEPS
- find out what other companies do
- Inform outsourcer that performance tests are no longer optional to complete by GA

Deployment Plan - AC (will know schedule by end of day Thu)
Satish
Melora
Lisa
Neal

Rollback Plan - AC (will start Thurs)
Rod
Rob
Jill
Stev

Load Test
---
Create Use Cases
Create Scripts
Review Load Test Results for  deployment criteria

## 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**<br><br>*Description:*<br>As a Windows 95 product, it should conform to the Win95 logo requirements and interface guidelines. | *ST Labs*<br>• 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**<br><br>*Description:*<br>The product should function in substantial conformance to the Product Requirements Specification and user documentation | *ST Labs & COMPANY*<br><br>• exploratory testing<br>• documentation-based testing<br>• specification-based testing<br>• scenario testing<br>• input domain testing |
| **HW Compatibility**<br><br>*Description:*<br>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*<br>• Verify that we have the config. info on each beta tester.<br><br>*ST Labs*<br>• (include configurations)<br>• Microphones<br>• Sound cards<br>• Systems<br><br>*COMPANY*<br>• (include configurations) |
| **Installability**<br><br>*Description:*<br>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.)<br><br>*ST Labs & COMPANY*<br>• monitor beta testers<br>• clean install testing<br>• upgrade install testing<br>• uninstall testing<br>• installing a new context |
| **SW Compatibility**<br><br>*Description:*<br>As a product to be deployed in an open environment, it must be operable with a variety of popular software products. | *ST Labs*<br>• Quick dictation interference test<br>  • Not enough time to test with NT network.<br>  • Applications that may also use SoundBlaster<br>  • Netscape<br>  • Exchange clients (MSMail, Schedule Plus)<br>  • SAM virus clinic<br>• Basic interoperability testing (there is not enough time to do comprehensive testing, here)<br>  • non-speech-aware dictation clients<br>  • notepad<br>  • Ami Pro<br>  • Word |

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

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

| | |
|---|---|
| **Memory/Mass Storage**<br><br>*Description:*<br>Users may experience failures associated with the large amount of internal memory and mass storage required for this product.<br><br>• *efficiency:* how files are stored and cleaned up.<br>• *reliability:* what happens under low memory or disk space conditions. Memory leaks.<br>• *usability:* how do users know when and how to delete files or optimize their systems. | *ST Labs*<br>• exploratory testing<br>• documentation-based testing<br>• documentation testing<br>• monitor beta testers<br>• all stress testing is the responsibility of COMPANY.<br><br>*COMPANY*<br>• stress testing<br>  • simultaneous applications low disk space and memory configs.<br>  • long dictation sessions<br>  • large number of corrections<br>  • add lots of words in a session |
| **Performance**<br><br>*Description:*<br>Because the usability of the system is strongly dependent on system performance, this performance should be measured and monitored.<br><br>• *Performance dimensions:*<br>  • during initial acoustic adaptation<br>  • during dictation<br>  • language model adaptation (short-term & long-term)<br>  • during further acoustic adaptation<br>• *Performance degradation (due to):*<br>  • lack of disk space or internal memory<br>  • dictation session duration<br>  • dictation file size<br>  • number of corrections<br>  • size of dictionary | *ST Labs*<br>• Qualitative performance testing of framework applications (MIP, unless critical)<br><br>*COMPANY*<br>• Performance testing of speech recognition technology |
| **Usability**<br><br>*Description:*<br>Users may find this product hard to learn and frustrating to use.<br><br>*Factors:*<br>• We know very little about the target market and typical user.<br>• The product requires that the user adopt a particular style of dictation.<br>• 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.<br>• The product consumes immense computing resources, particularly storage, and requires the use to perform housekeeping on a regular basis. | *Beta Testers*<br>• The beta testing process our primary means of assessing how much of a problem this is.<br><br>*ST Labs*<br>• Supporting the beta test process and pre-filter or summarize problems.<br>• Mention in passing any ideas or concerns about this problem.<br>• 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<br>http://www.adobe.com/newsfeatures/year2000/prodlist.html |
| **Dell Power Edge 6100** | Compliant | http://www.dell.com/year2000/faq/faq.htm<br>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 on 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.

# Appendix: Test Cases

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 Docs | ✓ | ✓ | | ✓ |
| 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 |
| US | 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 |

# 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**

# Round Results Risk Areas (v1)

<table>
<tr><th>Risk Area</th><th>Test Strategy & Tasks</th></tr>
<tr><td>

## Critical Calculated Items

*A critical calculated item may be wrong.*

Critical calculated items are data that meet three conditions:

1. They are calculated by the system (not preset in a database).
2. They are not easily determined to be right or wrong (the eyeball oracle is not reliable).
3. It will be a very bad problem if they are wrong. By bad problem we mean something that threatens the integrity of an auction, either by stopping the auction or impairing its ability to maximize profits.

## Results Processing

*Round results processing may fail in a way that interrupts the auction.*

Round results processing includes the events that occur between rounds that assess the status of the auction and prepare for the next round of that auction. It encompasses those aspects of the auction administration GUI that control the processing tasks.

## Parameter Changes

*Changing an auction parameter may fail to change the auction, or may change it in an undesired way, or may cause other failures that interrupt the auction.*

Parameter changes are any modification of the auction rules between rounds. Parameter changes affect future rounds of the current auction. This risk area encompasses all aspects of the auction administration GUI that involve parameter display and change.

## Reports

*An auction report may be missing critical information, or critical information it presents could be incorrect.*

The reports risk encompasses the quality of the content of each report.

</td><td>

- ❑ For each of these risk areas, create a separate test coverage outline. In the case of CCI's, itemize each one, and produce a short outline for each that includes the conditions that might affect that CCI.
- ❑ For each risk area, flesh out a list of specific risks that belong to that area.
- ❑ Determine one or more test activities for each risk area that covers a requisite variety of conditions that might affect that area (from the TCO), and/or covers a risk in the list of specific risks for that area.
- ❑ Consider tools and testability enhancements that will make it easier to test.
- ❑ Design a smoke test activity for each area. This is a quick test of each area can stand in for the full suite of tests in the event there is not enough time to fulfill the complete test strategy (such as after a quick fix at the last minute).
- ❑ Many tests for round results will require specifying auction setup, bidding scenarios, and oracles to be used at the end of each round.
- ❑ All other things being equal, the test activities for each area should include testing for capability (can each function work?), reliability under *foreseeable* conditions and combinations of conditions, and reliability under *realistic* conditions. The depth of testing should reflect the importance of the functions and conditions involved.

</td></tr>
<tr><td>

## Preset Auction Parameters

*Preset auction parameters may be incorrect in a way that causes the auction to fail, or to succeed less well than it otherwise would have.*

Preset auction parameters are values stored in the auction database or INI file that control the scope of and behavior of an auction. These include things like license data and round definitions.

</td><td>

[Is this within the scope of our testing? If so, how do we test it?]

</td></tr>
<tr><td>

## Platform Configuration

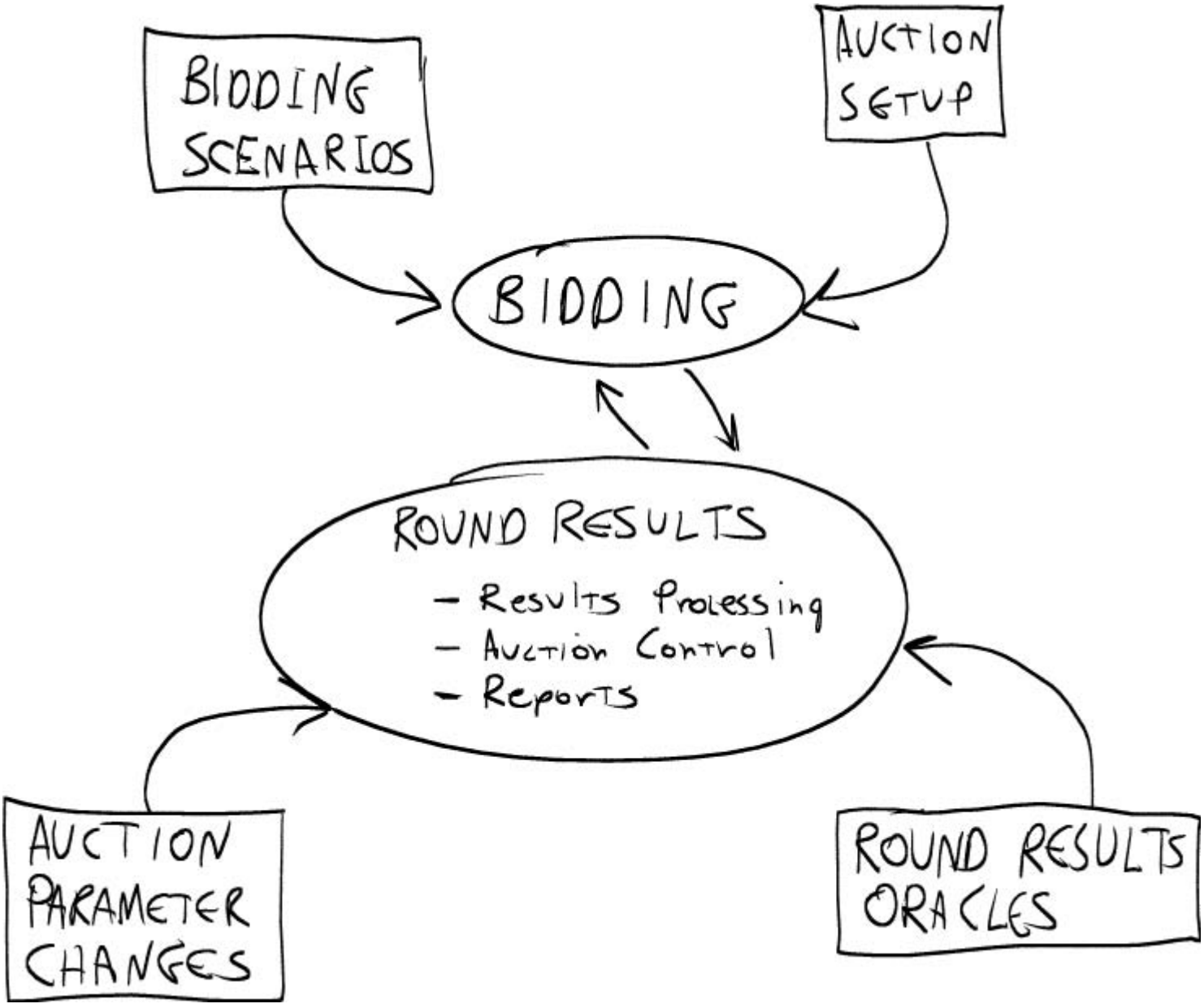
*A misconfigured server could cause the auction to fail, or it could block testing or lead to false bug reports. A mis-configured client*

Platform configuration includes the files and environment variables on the servers, and the configuration of the auction administrator client. Is the right product installed? Is it installed correctly?

</td><td>

- ❑ Monitor file configuration with CCTEST tool.
- ❑ When any changes are made to the configuration, determine if any testing is needed.

</td></tr>
</table>


BIDDING SCENARIOS
AUCTION SETUP
BIDDING
ROUND RESULTS
- Results Processing
- Auction Control
- Reports
AUCTION PARAMETER CHANGES
ROUND RESULTS ORACLES

# Exploratory Testing Session Sheets

*The following few pages are* session sheets*—notes from several exploratory testing sessions for a released commercial product called DecideRight. The purpose of the program is to help the user to make better decisions. The user enters the options to be considered, the criteria upon which the decision will be made, and the weight or significance of each criterion. The program then evaluates the data and presents a recommended decision.*

*The test notes, bug reports, issues, and other data on the session sheet supplement the debriefing that happens between the tester and the test manager or test lead, typically performed just after the session has ended. In later test cycles, the session sheet can be used to guide checks for bug fixes and regression tests.*

*The session sheet is in a structured format that can be scanned by a Perl program that parses the data and compiles coverage information into an Excel spreadsheet.*

*The testers prepared these session sheets during and immediately after each session. Note the progression of the sessions; the first two are from the first day of testing, in which the focus is on exploring the product, building models of the test space, identifying coverage areas and oracles, and identifying issues that could threaten the value of the testing project—testing to learn. The third and fourth sheets are from the afternoon of the second day, in which the focus is more strongly oriented towards finding problems in the product—testing to search. In all sessions though, both searching and learning happens.*

*For more examples of session sheets, the tools to scan them, and more detailed information on Session-Based Test Management, see http://www.satisfice.com/sbtm.*

CHARTER
---
Use the heuristic test design model to devise a DecideRight test strategy.

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

START
---
4/16/01 9:30am

TESTER
---
Jonathan Bach

TASK BREAKDOWN
---

#DURATION
normal

#TEST DESIGN AND EXECUTION
80

#BUG INVESTIGATION AND REPORTING
0

#SESSION SETUP
20

#CHARTER VS. OPPORTUNITY
100/0

DATA FILES
---
#N/A

TEST NOTES
---
The major purpose of DecideRight is to help make difficult, high-stakes decisions. Therefore, our primary concern in testing it is to evaluate the correctness of decisions that it suggests, and the ability of users to properly operate the product to obtain those decisions. Although we will focus the bulk of our effort on those risk areas, we will also spend some time testing the general functionality of the product.

Test strategy:

* Understand the decision algorithm and generate a parallel decision analyzer using Perl or Excel that will function as a reference oracle for high volume testing of the app.

* Create a means to generate and apply large numbers of decision scenarios to the product. This will be done either through the use of a GUI test automation system, if practical, or through a special test facility built into the product (if development is able to provide that), or through the direct generation of DecideRight scenario files that would be loaded into the product during test.

* Review the documentation, and the design of the user interface and functionality for its sensitivity to user error that could result in a reasonable misunderstanding of decision parameters, analysis, or suggestions.

* Test with decision scenarios that are near the limit of complexity allowed by the product. (We will investigate creating these scenarios automatically.)

* Compare complex scenarios (Automatically, if practical).

* Test the product for the risk of silent failures or corruptions in the decision analysis.

* Using requirements documentation, user documentation, or by exploring the product, we will create an outline of product elements and use that to guide user-level capability and reliability testing of the product.

BUGS
---
#N/A

ISSUES
---
#ISSUE
The decision algorithm is difficult to understand and simulate.

#ISSUE
Risk of coincidental failure of both the simulation and the product.

#ISSUE
Automating decision tests will be difficult.

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

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.

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

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
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 ?

CHARTER
---
Using the steps outlined in the manual, create a decision table manually noting any significant differences than when using QuickBuild.

#AREAS
OS | Win98
Build | 1.2
DecideRight | Main Table window
Strategy | Complex | Stress Testing
Strategy | Complex | Function & Data Testing

START
---
4/17/01 5:30pm

TESTER
---
Jonathan Bach
Tim Parkman

TASK BREAKDOWN
---

#DURATION
normal

#TEST DESIGN AND EXECUTION
50

#BUG INVESTIGATION AND REPORTING
30

#SESSION SETUP
20

#CHARTER VS. OPPORTUNITY
90/10

DATA FILES
---
Thursday.drd

TEST NOTES
---
Stepped through the steps in the manual, starting on page 5-1 to walk through a new decision table -- ended on page 5-6

  * Clicked toolbar buttons (see BUG 1)
  * added options and criteria w / weighting (see BUG 8)
  * added options that were non-alphenumeric characters
  * tested Optional Overview field -- 32000 characters entered
  * Edit menu: Add Option (see BUG 2) (via menu and clicking.  Tested how many options you can list and alphabetizing (see BUG 3 and 7)
??? Manual says that I can return to the table by "clicking any other table element."  Not sure what this means.
  * Entered a description for an option
  * Changed the name of an existing option
  * Added a new option (see BUG 4 and 5)
  * Verified a option can be deleted with right-click menu or edit menu
??? Should Undo work after deleting an option?  It doesn't. (see ISSUE 2)
  * Added 63 columns of criteria -- (see BUG 6 & ISSUE 1)

OPPORTUNITY: tested Find/Replace on option description (DCR about no Replace button -- UI shows a confirm instead) -- spent about 10 minutes testing the max length of the description field

After creating a table using the steps in the manual, I didn't see any important differences from using QuickBuild.

BUGS
---
#BUG 1
UI: paper clip and pushpin buttons are not disabled, even though they do nothing

Repro:
1 -- create a new decision manually
2 -- click either paper clip or pushpin button (fly out text says "view/edit documents that explain a decision element"

Result: No response.

Expected: Should be grayed out, else to perform the function that the flyout menu claims should be performed.

#BUG 2
No accelerator keys for some menu items

Repro:
The following menu items do not have underbars:
File | Preferences
Edit | Add Option
Edit | Add Criterion
Edit | Delete Option
Edit | Delete Criterion
Edit | Find/Replace
Edit | Numeric Values
Edit | Optional Epxlanation
Edit | Documents
View | Ratings Graph
View | Baseline Comparison Graph
View | Previous Criterion
View | Next Criterion
View | Previous Option
View | Next Option
Format | Recalc Disable (Minimize Table)

#BUG 3
Decision table lets you have options and criteria that are identically named

Repro:
1 -- Make a decision table manually
2 -- add two criteria and options with the same name

Result: no warning that there is a duplicate
Expected: No duplicates to be allowed, because of potential confusion to user

#BUG 4
When focus is on Option list in table, there is no arrow key support to scroll through list

Repro:
1 -- make a new decision table
2 -- highlight an option
3 -- press the up or down arrows

Result: No response.
Expected: Arrow keys should be active to scroll through the list of options.

#BUG 5
Data can't be entered in entry box for new option when focus is on table

Repro:
1 -- In the option view, click on the table
2 -- click on the insertion point
3 -- type something

Result: There is an insertion-point cursor, but keyboard is unresponsive.

#BUG 6
Crash -- GPF in DECIDER.EXE (crash in GDI.EXE in module 00016:000007f1 when entering over 60 columns of criteria

Repro:
1 -- create a new decision table (manually)
2 -- add criteria by typing in a name and hitting ENTER
3 -- repeat approx. 60 columns

Result: GPF in GDI.EXE.  When you try to launch DecideRight again, dialog pops up -- "not enough free memory" even though there is no other app open.

#BUG 7
New option added to existing options does not get sorted alphabetically

Repro:
1 -- create a new decision table
2 -- add 5 or 6 options
3 -- add some criteria
4 -- go back and add another option

Result: The new additional option does not get alphabetically sorted like the others until the table is closed and re-opened.

#BUG 8
Identical options can be entered

ISSUES
---
#ISSUE 1
Is there a recommended maximum number of criteria?  We were getting GPFs with about 60 columns.

#ISSUE 2
Should Undo work after deleting an option?  It doesn't.

**Variables in Page Setup Dialog Box for Powerpoint:**

Slide-sized-for
- On-Screen Show
- Letter Paper
- Ledger Paper
- A3
- A4
- B4
- B5
- 35mm
- Overhead
- Banner
- Custom (activated automatically when w/h changed)

- ? fills in height and width differently depending on printer?

Height/Width
- Automatic setting from slide-sized-for
- Automatic setting from portrait-landscape
- Min (1)
- Max (56)
- Increment (by .1; also rounds to nearest .1)
- Precision (hundredths)
- Size (32 characters max; 0 min.)

- Vs. print preview
- Vs. size of fonts on page (font sizes on page are automatically scaled)
- Vs. wordart and other objects on the page

Number-slides-from
- Max (9999)
- Min. (0)
- Representatives
  - 1
  - any two, three, and four digit number
  - any number that when added to number of slides printed exceeds 9999

- Vs. number of slides in presentation

Slide-orientation
- Automatic setting from w/h

Notes-orientation
- ? Not automatically set from w/h. Why not?

Context help
    Question Mark icon

General Edit Field Functionality
    Up/Down arrow
    Left/Right arrow
    Home/End
    Copy/Cut/Paste

**Sources of information:**

- exploratory testing
- question mark icon
- help file

| Variable | Equivalence classes | Risk factors |
|---|---|---|
| Slide-Sized-for | - On-Screen Show<br>- Letter Paper<br>- Ledger Paper<br>- A3<br>- A4<br>- B4<br>- B5<br>- 35mm<br>- Overhead<br>- Banner<br>- Custom | - Size larger than printable area<br>- Size smaller than printable area<br>- Size exactly printable area<br>- Portrait vs. Landscape |
| Height/Width | - 1 (min)<br>- 56 (max)<br>-<br>- Increment (by .1; also rounds to nearest .1)<br>- Precision (hundredths)<br>- Size (32 characters max; 0 min.)<br>-<br>- Vs. print preview<br>- Vs. size of fonts on page (font sizes on page are automatically scaled)<br>- Vs. wordart and other objects on the page<br>- | - Size larger than printable area<br>- |

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?*

## Scenario Themes

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.<br>- 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.<br>- 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.<br>- 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.<br>- Tasks should have *variety*, e.g. short ones, long ones, key tasks, non-key tasks, started, not-started, with and without attachments and checklists.<br>- Set the simulation date to intersect with the project data that you are using.<br>- 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*:<br>❑ Select a scenario test charter and spend about 90 minutes testing in accordance with it.<br>❑ Perform the activities described in the test charter, but also perform variations of them, and vary the sequence of your operations.<br>❑ 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.<br>❑ 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.<br>❑ Do things that should cause error messages, as well as things that should not.<br>❑ Ask questions about the product and let them flavor your testing: What will happen if I do *this?* Can the product handle *that?*<br>❑ Consider working with more than one tester on more than one scenario. Perform multiple scenarios together.<br>❑ 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.<br>- Review and apply the HICCUPP heuristics.<br>- For each operation that you witness the product perform, ask yourself how you know that it worked correctly.<br>- Perform some operations with data chosen to make it easy to tell if the product gave correct output.<br>- 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.<br>- Make a note of obstacles you encountered in the test process itself.<br>- 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)<br>❑ 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.<br>❑ For some tasks, view details, checklists, and attachments.<br>❑ Update each task in some way, including:<br>- No update<br>- "Mark as Updated"<br>- Shorten duration remaining<br>- Set remaining duration to zero; or "Mark as Completed"<br>- Increase duration remaining<br>- Provide comments; update checklist<br>- Undo some updates<br>❑ 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.<br>- View updated tasks *after buffer update* to verify they are correct.<br>- 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.<br>- 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.<br>- *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.<br>- *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.<br>- *INTERRUPTION:* Try to make updates while a buffer update is going on.<br>- *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.<br>- Buffer consumption for one of the projects should ideally be in the yellow or red.<br>- 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.<br>❑ 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.<br>❑ View all impact chains for your project, and for some of those tasks:<br>- view task details<br>- view task links<br>- view task load chart<br>❑ 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.<br>❑ 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.<br>❑ 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.<br>- View updated tasks *after buffer update* to verify they are correct.<br>- 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.<br>- Determine the total number of tasks visible within MS project file, and verify all are visible in Enterprise.<br>- 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.<br>- *TUG OF WAR:* a second user logs in and checks out the project that you are analyzing, locking it.<br>- *OOPS:* update project notes and comments in the wrong project, and try to remove them and apply them to the right project.<br>- *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.<br>❑ Update the project network in MSP, do a selection of the following:<br>- Add new tasks that have starting dates before the present date, some that span the present date, and some that end in the future.<br>- Add new tasks that are not on the critical chain, and some that are.<br>- Delete some tasks.<br>- Modify data in custom fields.<br>- Change some of the task linkages.<br>- Reassign resources; Overload some resources.<br>- If the project has one endpoint, add a second endpoint; if it has two multiple endpoints, remove all but one.<br>*(remember to keep track of the changes you make!)*<br>❑ Check the project back into PCE, and update buffers.<br>❑ View all impact chains for your project, and for the tasks and chains that you modified:<br>- view task details<br>- view task links<br>- view task load chart |
| Oracle Notes | - The new network's info are correctly represented in PCE:<br>- buffer consumption<br>- impact chain<br>- key tasks<br>- resources and managers<br>- 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.<br>- *OOPS:* Check in the wrong project file, and then try to recover.<br>- *OBJECT LIFECYCLE:* Rewire the project several times, interspersing that with UP1 an UP2 scenarios. Then complete all tasks. |

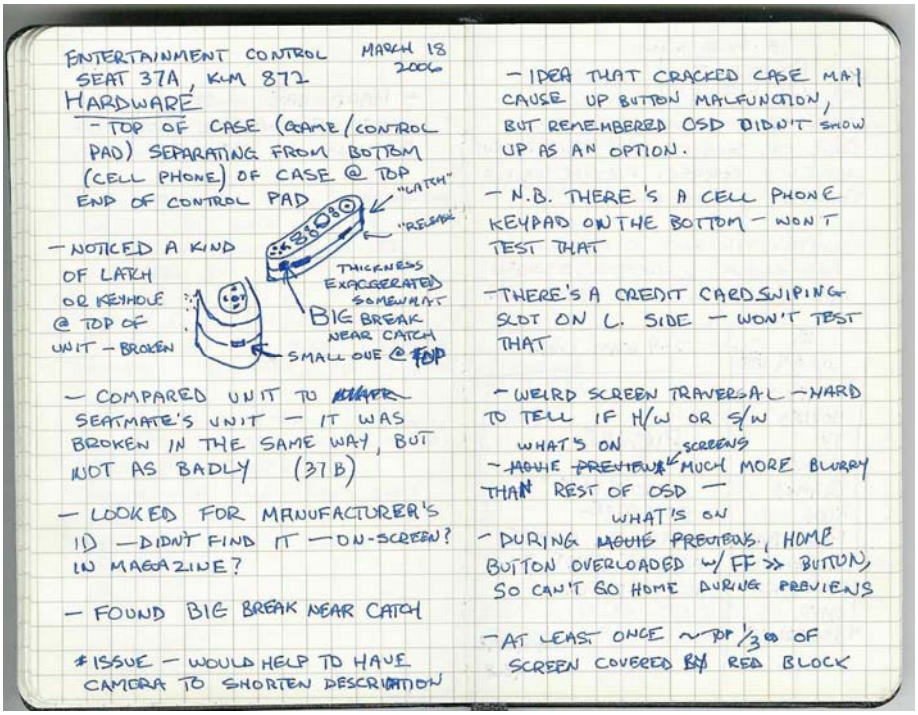
## *Notes from an Exploratory Testing Session*

By Michael Bolton

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.

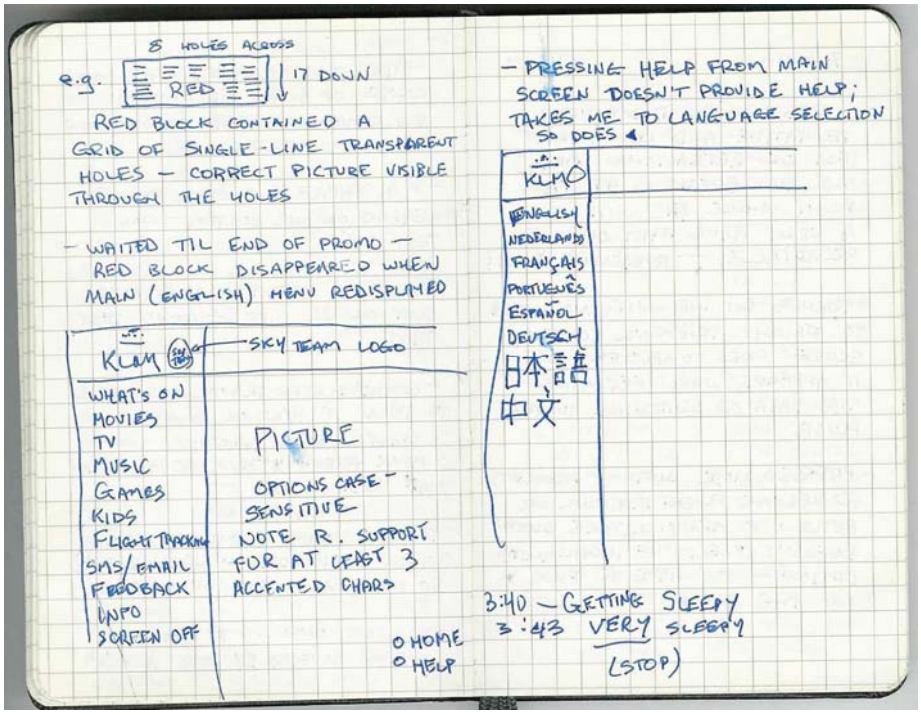
The system was menu-driven. I went to the page that listed the movies that were available, and after scrolling around a bit, I found that the "Up" button on the controller didn't work. I then inspected the controller unit, and found that it was cracked in a couple of places. Both of the cracks were associated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



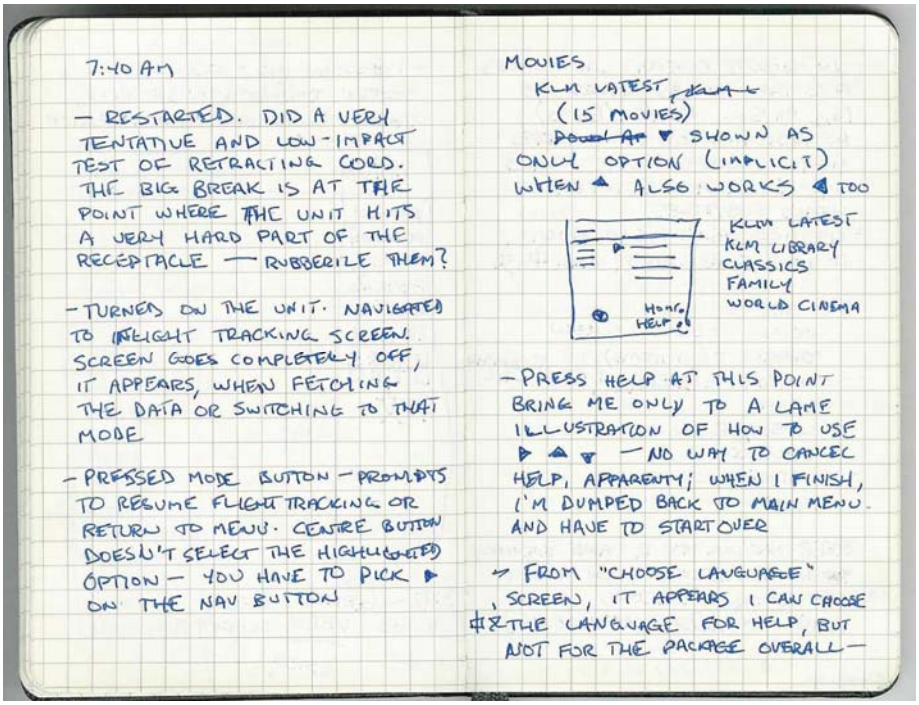

When I take notes like this, they're a tool, not a product. I don't expect to show them to anyone else; it's a possibility, but the principal purposes are to allow me to remember what I did and what I found, to guide a discussion about it with someone who's interested, or to help with planning and strategizing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

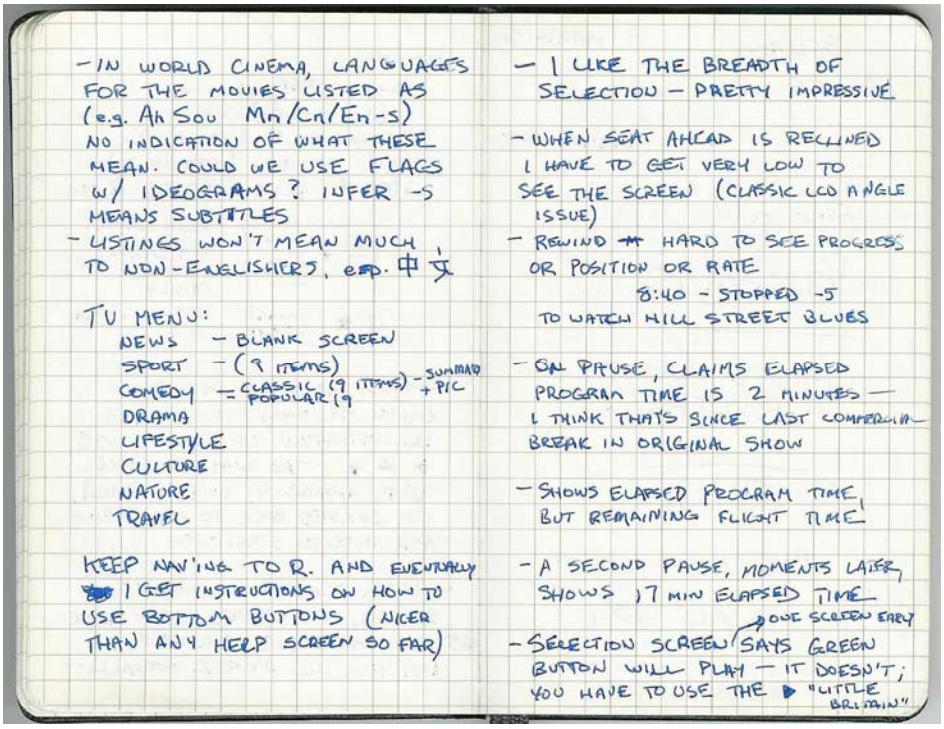
I don't draw well, but I'm slowly getting better at sketching with some practice. I find that I can sketch better when I'm willing to tolerate mistakes.




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 state, they might suggest areas that I should revisit later.






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 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? A notebook is inexpensive, lightweight, portable, low cost, and high value. You might lose it, but it never crashes. If anything is missing from the notes, there's a very high probability that they'll still help me enough to reminder specific details, even months after the fact. The test 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.

# 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.

## File and Path Name Test Matrix

Company Name:
Author or maintainer:
Product / Release under test:

Purpose: The purpose of this table is to provide a set of test cases for filename and pathname handling under Windows 9x and the Windows NT family. Both valid and invalid test cases are included.

Notes: Cell formatting within the matrix colours the cell green when the value P (for Pass) is entered; red with F (for Fail); orange with W (for Warn)

Tests of invalid input should ensure not only that error messages are displayed, but are displayed appropriately.

Note that this set of tests is intended to apply to the handling of the filename only; other test matrices deal with actual file input and output.
Include new tests as you devise them, or as found problems inspire them.

| Valid Filenames | Win95 | Win98 | WinME | WinNT 3.5 | WinNT 4.0 | WinNT 4.0 SP6 | Win2K | Win2K SP1 | Win2K SP3 | WinXP | WinXP SP1 | WinXP SP2 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8.3 Format | | | | | | | | | | | | | | | | | | | | |
| LFN | | | | | | | | | | | | | | | | | | | | |
| LFN with spaces | | | | | | | | | | | | | | | | | | | | |
| LFN file with LFN path | | | | | | | | | | | | | | | | | | | | |
| No extension (added by program) | | | | | | | | | | | | | | | | | | | | |
| Path and file names that include numbers | | | | | | | | | | | | | | | | | | | | |
| Unusual but valid characters (!@#$%^&-_=) | | | | | | | | | | | | | | | | | | | | |
| Invalid DOS (but valid Windows) characters (+;) | | | | | | | | | | | | | | | | | | | | |
| Valid UNC path | | | | | | | | | | | | | | | | | | | | |
| All spaces for extension | | | | | | | | | | | | | | | | | | | | |
| Filename containing periods before the extension | | | | | | | | | | | | | | | | | | | | |
| Pathname containing periods before the extension | | | | | | | | | | | | | | | | | | | | |
| Handle filename for file that already exists | | | | | | | | | | | | | | | | | | | | |
| Handle filename for file that does not yet exist | | | | | | | | | | | | | | | | | | | | |

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Invalid filenames** | | | | | | | | | | | | | | | | | | | | |
| No filename | | | | | | | | | | | | | | | | | | | | |
| Non-spec file extension | | | | | | | | | | | | | | | | | | | | |
| Invalid characters (< > / \ \| : " * ?) in filename | | | | | | | | | | | | | | | | | | | | |
| Invalid characters (< > / \| " * ?) in path | | | | | | | | | | | | | | | | | | | | |
| Valid characters in path, but in wrong position (e.g. D\FOO.TXT) | | | | | | | | | | | | | | | | | | | | |
| Leading space (trimmed properly?) | | | | | | | | | | | | | | | | | | | | |
| Trailing spaces (trimmed properly?) | | | | | | | | | | | | | | | | | | | | |
| Valid filename, invalid path | | | | | | | | | | | | | | | | | | | | |
| Valid path; invalid filename | | | | | | | | | | | | | | | | | | | | |
| Invalid drive spec | | | | | | | | | | | | | | | | | | | | |
| \\Volume\Share\Path\File with single starting backslash | | | | | | | | | | | | | | | | | | | | |
| Double backslash anywhere except first two characters | | | | | | | | | | | | | | | | | | | | |
| Triple starting backslash | | | | | | | | | | | | | | | | | | | | |
| Forward slashes instead of backslashes | | | | | | | | | | | | | | | | | | | | |
| All spaces for filename | | | | | | | | | | | | | | | | | | | | |
| Filename longer than MAX_PATH | | | | | | | | | | | | | | | | | | | | |
| Pathname longer than MAX_PATH | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| **Additional Test Ideas** | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | |

## *Rapid Software Testing Reading, Resources and Tools*

Compiled by Michael Bolton and James Bach
Last revised January 4, 2007

To learn about **finding bugs and risks**, read

- *Lessons Learned in Software Testing* by Cem Kaner, James Bach, and Bret Pettichord. 293 bite-sized lessons from three of the leaders of the Context-Driven School of Software Testing.
- *Testing Computer Software* by Cem Kaner, Jack Falk, and Hung Quoc Nguyen. The book that, for many testers, started it all. The best-selling testing book in history. Somewhat out of date these days, since it predates the rise of Windows and the rise of the Internet, but a very important text in terms of the *thinking* part of software testing. Also contains an excellent (if overwhelming) example of a bug and testing taxonomy.
- *How to Break Software* by Whittaker, and *How to Break Software Security* by Whittaker and Thompson. Two wonderful testing books that actually (gasp!) show specific bugs and the classes of tests that expose them. This book presents a useful perspective for finding problems—identifying customers of the application as the end-user, the file system, the operating system, and application programming interfaces.
- *Hacking Exposed* by Stuart McClure, Joel Scambray, and George Kurtz, and *Hacking Web Applications Exposed* by Joel Scambray and Mike Shema. Hackers and testers have a lot in common in terms of the approaches that they can use to find out how software really works and how to expose its weaknesses. Testers owe hackers a favour, in a way, since the work of the former represents risk that underscores the value of the latter.

To learn about **testing philosophy**, read

- *The Pleasure of Finding Things Out* by Richard Feynman. In particular, read his Appendix to the Rogers Commission's report on the Challenger.
- *Surely You're Joking, Dr. Feynman! Adventures of a Curious Character* by Richard Feynman. Feynman's curiosity drove his apparently insatiable desire to find out about the world, often in the same manner that a tester or hacker might. This book contains (among other things) accounts of Feynman's safecracking exploits at Los Alamos.
- *What Do You Care What Other People Think?* by Richard Feynman. The first page of this book alone—in which Feynman notes that learning about things only adds to a deeper appreciation of them—would make it a worthwhile recommendation.
- *Introduction to General Systems Thinking* by Jerry Weinberg
- *Are Your Lights On* by Don Gause and Jerry Weinberg. A more lightweight approach to some of the concepts in the latter book.
- *Quality Software Management Vols. 1 – 4* by Jerry Weinberg. Lots of different angles on software quality from one of the patron saints of software testers.
- *Anything* by Jerry Weinberg

To find good stuff **on the Web** about testing and other topics, see

- Black Box Software Testing Course (http://www.satisfice.com/moodle) This course was co-authored by Cem Kaner and James Bach, and contains much in common with Rapid Software Testing. The course features video lectures, course notes, recommended readings, self-study and self-testing resources. Comprehensive—and free.

- Cem Kaner (http://www.kaner.com) An overwhelming collection of articles, papers, and presentations on software testing, test management, elements of software law.
- James Bach (http://www.satisfice.com) A less overwhelming but still comprehensive collection of essays, papers, and tools from the author of the Rapid Software Testing course.
- Michael Bolton (http://www.developsense.com) Articles and resources on software testing topics, including test matrices, all-pairs testing, installation programs, and beta tests. Also refer to the archived newsletters.
- The Florida Institute of Technology (http://www.testingeducation.org) The host for the Black Box Software Testing course above, this site also contains a large number of interesting links and articles, many written and produced by Cem Kaner and his students at Florida Tech.
- Risks Digest (http://catless.ncl.ac.uk/risks) A fine collection of horror stories and risk ideas that makes for excellent occasional browsing.
- StickyMinds (http://www.StickyMinds.com) The online presence for Better Software magazine (formerly Software Testing and Quality Engineering; STQE; "sticky"—get it?). There's a big collection of articles here of varying value. Articles from the magazine and "StickyMinds Originals" have been edited and tend to be of higher quality than the contributed articles.
- For tutorials on various markup languages, browser scripting, server scripting, and technologies related to Web development, try www.w3schools.com.

To learn **other wonderful stuff that I believe is worth thinking about**, look at

- *Please Understand Me* by David Kiersey ◆ The Myers-Briggs Type Inventory, which provides insight into your own preferences and why *other people* seem to think so strangely.
- *The Visual Display of Quantitative Information*, Edward Tufte ◆ How to present information in persuasive, compelling, and beautiful ways. Other books by Tufte are terrific, too—and if you ever have an opportunity to attend his one-day course on presentation, do it!
- *A Pattern Language*, Christopher Alexander ◆ A book about architecture, even more interesting as a book about thinking and creating similar but unique things—like computer programs and tests for them.
- *Domain Driven Design* by Eric Evans, in which he introduces the concepts of "ubiquitous language"—essentially making sure that everyone in the project community is using the same terms to describe the project domain, even to the extent that that language is used in the code itself; and "knowledge crunching"—essentially why all those meetings and documents and diagrams and discussions are valuable, and how they can become more effective.
- *Better Software*, a most unfortunate name of an otherwise wonderful magazine. Excellent information for professional testers. Michael writes a monthly column for this magazine.
- *The Amplifying Your Effectiveness Conference*, held every November in Phoenix, hosted by Jerry Weinberg and his colleagues. AZ. See http://www.ayeconference.com for details.
- *Blink*, by Malcolm Gladwell. A pop-science book about rapid cognition. There are four central points that he tries to express in the book: 1) Snap judgments are a central part of how we make sense of the world. 2) Snap judgments are vulnerable to corruption by forces that are outside of our awareness. 3) It's possible that we may improve our snap judgments by removing information. 4) Instead of solving the problem by fixing the decision-maker, change the context in which the decision is made. Note that the book doesn't always make it clear that these are the points; I got these from attending a lecture by Mr. Gladwell during the book tour, in which he addressed some of the criticisms of the book with these four points. Another point that came up during the lecture: experts simplify the field in front of

them, because of their expertise behind them. In the moment, they whittle down to the essentials.

Mr. Gladwell's other work—his book *The Tipping Point*, and his New Yorker articles, archived at http://www.gladwell.com —is informed by the idea that little things make a big difference. Not all of it can be directly related to testing, but it's all fun reading.

- *About Face: The Essentials of User Interface Design* and *The Inmates are Running the Asylum: Why High Tech Products Drive Us Crazy and How To Restore The Sanity*, by Alan Cooper. In both of these books, Mr. Cooper provides some interesting and thoughtful (and sometimes provocative) tips for people involved with the design of computer software. Most software shows us a map of its own internals, and the user interface (or, as Mr. Cooper calls it, the user interaction) must be adapted to fit that functional map. Yet customers purchase software to get work done; Mr. Cooper consistently and expertly advocates keeping the user's task in mind, and designing software that helps users instead of frustrating them. While both books are primarily oriented towards developers and software designers, managers and marketers should seriously consider reading both books, and particularly *Inmates*.

- *Code : The Hidden Language of Computer Hardware and Software*, by Charles Petzold. *Code* is about encoding systems--the kinds of systems by which we represent numbers and letters using computers and other kinds of machines. Hmm.... encoding systems. Sounds fascinating, huh? As a matter of fact, this is a highly useful book. I wish it had been around when I was learning about computing machines; the book would have made a lot of things clear right away. Effective testers need to know something about boundary conditions; so do effective programmers. Numbers like 255, -32768, 65335, 4294967295, and -2147483648 are interesting; so are symbols like @, [, `, and {. Don't know why? *Code* will tell you.

  The book helps the reader to understand some of the otherwise obscure boundary conditions that exist because of the ways that computers work, and because of the choices that we've made in constructing those machines. You also get to understand what those dots of Braille mean, and how machines (under our instructions, of course) make decisions and evaluate information. This book probably isn't for everyone, but anyone on the engineering side of the computer community should be familiar with the principles that Mr. Petzold explains so clearly.

- *Tools of Critical Thinking*, by David A. Levy, 1997. This is a key book for Rapid Testers, in that it provides terrific, digestible descriptions of "metathoughts"—ways of thinking about thinking, and in particular, thinking errors and biases to which people are prone. This book purports to be about clinical psychology, but we think it's about about the thinking side of testing in disguise.

- *Exploring Requirements: Quality Before Design*, by Don Gause and Gerald M. Weinberg

- *How to Solve It*, by George Polya.

- *Cognition in the Wild*, by Edwin Hutchins

- *Thinking and Deciding*, by Jonathan Baron

- *Lateral Thinking: Creativity Step by Step*, Ed De Bono
- *The Social Life of Information*, John Seely Brown, Paul Duguid
- *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*, by Donald Norman
- *The Sciences of the Artificial, 3rd Ed.*, by Herbert A. Simon
- *Conjectures and Refutations: The Growth of Scientific Knowledge*, by Karl Popper
- *Theory and Evidence: The Development of Scientific Reasoning*, by Barbara Koslowski
- *Abductive Inference: Computation, Philosophy, Technology*, by John R. Josephson and Susan G. Josephson
- *Science as a Questioning Process*, by Nigel Sanitt
- *Administrative Behavior, 4th ed.*, by Herbert Simon
- *Software Testing: A Craftman's Approach*, by Paul C. Jorgensen
- *Bad Software: What to Do When Software Fails*, by Cem Kaner and David Pels
- *Introducing Semiotics*, by Paul Cobley and Litza Jansz
- *Proofs and Refutations*, by Imre Lakatos
- *Play as Exploratory Learning*, by Mary Reilly
- *Radical Constructivism: A Way of Learning (Studies in Mathematics Education)*, by Ernst von E. Glasersfeld
- *Why Art Cannot Be Taught*, by James Elkins
- *Exploratory Research in the Social Sciences*, by Robert A. Stebbins
- *Applications of Case Study Research*, by Robert K. Yin
- *What If....Collected Thought Experiments in Philosophy*, by Peg Tittle
- *Abductive Inference : Computation, Philosophy, Technology*, by John R. Josephson
- *Time Pressure and Stress in Human Judgment and Decision Making*, edited by A.J. Maule and O. Svenson

- *Outlines of Scepticism*, by Sextus Empiricus

- *System of Logic Ratiocinative and Inductive*, by John Stuart Mill

## Tools

The simplest way to find these tools, at the moment, is to Google for them. Everything listed here is either free or a free trial; we encourage readers to register the commercial products if you find them useful.

In addition to the tools listed here, check out the tools listed in the course notes and in the article "Boosting Your Testing Superpowers" in the Appendix. Danny Faught also provides reviews and listings of testing and configuration management tools at http://www.tejasconsulting.com/open-testware/.

**Netcat** (a.k.a. NC.EXE) This is a fantastic little tool that, from the command line, allows you to make TCP/IP connections and observe traffic on specific ports. For lots of examples on how to use it, see the above-referenced Hacking Web Applications Exposed.

**SysInternals Tools** at http://www.sysinternals.com. These wonderful, free tools for Windows are probes that reveal things that we would not ordinarily see. **FileMon** watches the file system for opening, closing, reading, and writing, and identifies which process was responsible for each action. **RegMon** does the same thing for the Windows Registry. **Process Explorer** identifies which files are open and which Dynamic Link Libraries (DLLs) are in use by applications and processes on the system. **Strings** is a bog-simple little utility that dumps the textual contents of any kind of file, most useful for executables. I've found lots of silly little spelling errors with this tool; I've also found hints about the relationships between library files.

**Perl**. Grab Perl from the ActiveState distribution, http://www.activestate.com. They also have development tools that allow you to do things like create and distribute .EXE files from Perl scripts—which means that people can run programs written in Perl without having to install the whole gorilla. Also see CPAN, the Comprehensive Perl Archive Network at http://www.cpan.org. This is a library of contributions to the Perl community. Many, many problems that you'll encounter will already have a solution posted in CPAN.

**Ruby**. Get Ruby from www.rubycentral.com and/or the sites that link from it. After you've done that, look into the beginner's tutorial at http://pine.fm/LearnToProgram/?Chapter=00; some of Brian Marick's scripting for testers work at http://www.visibleworkings.com/little-ruby/. Then read the Pickaxe book whose real name is Programming Ruby (look up Pickaxe on Google); you might also like to look at the *very* eccentric "Why's Poignant Guide to Ruby" at http://poignantguide.net/ruby/.

**WATIR** (Web Application Testing In Ruby) and **SYSTIR** (System Testing In Ruby) are emerging and interesting tools based on Ruby, with the goal of permitting business or domain experts to comprehend examples or tests.

**SAMIE** was the Perl-based tool that at least partially inspired Ruby-based WATIR. SAMIE, with the Slingshot utility package, allows you to identify objects on a Web page so that you can more easily build a Perl-based script to fill out forms and drive the page.

**SnagIt**, a wonderful screen capture and logging utility from TechSmith. Available in trialware at http://www.techsmith.com

**TextPad**, a terrific text editor with excellent regular expression support and the ability to mark and copy text by columns as well as by lines. Shareware, available at http://www.textpad.com.

**PerlClip**, a utility for blasting lots of patterned data onto the Windows Clipboard for input constraint attacks. So-called because it's written in Perl, and it uses Perl-like syntax to create the patterns. Counterstrings—strings that report on their own length—are perhaps the coolest of several cool features. Written by James Bach and Danny Faught, and available free from http://www.satisfice.com and in the course materials for the Rapid Software Testing course.

**AllPairs**, to generate minimally-size tables of data that include each pair of possible combinations at least once. Written by James Bach and available free from http://www.satisfice.com and in the course materials for the Rapid Software Testing course.