



# **Exploratory Testing**

Michael Bolton  
DevelopSense  
<http://www.developsense.com>  
September 2009

## What IS Exploratory Testing?



- *Simultaneous test design, test execution, and learning.*

• *James Bach, 1995*

But maybe it would be a good idea to underscore why that's important...

## What IS Exploratory Testing?



- *Simultaneous test design, test execution, and learning, with an emphasis on learning.*
  - *Gem Kaner, 2005*

But maybe it would be a good idea to be really explicit about what goes on...

# What IS Exploratory Testing?



- I follow (and to some degree contributed to) Kaner's definition, which was refined over several peer conferences through 2007:

Exploratory software testing is...

- a style of software testing
- that emphasizes the personal freedom and responsibility of the individual tester
- to continually optimize the value of his or her work
- by treating test design, test execution, test result interpretation, and test-related learning
- as mutually supportive activities
- that run in parallel
- throughout the project.

So maybe it would be a good idea to keep it brief most of the time...

See Kaner, "Exploratory Testing After 23 Years",  
[www.kaner.com/pdfs/ETat23.pdf](http://www.kaner.com/pdfs/ETat23.pdf)

# Testing Isn't Just *Checking*

- Checking is a process of confirming and verifying existing beliefs
  - Checking can (and I argue, largely should) be done mechanically
  - It is a *non-sapient* process



See <http://www.developsense.com/2009/08/testing-vs-checking.html>

## What *IS* Checking?



- A *check* has three attributes
  - It requires an *observation*
  - The observation is linked to a *decision rule*
  - The observation and the rule can be applied

**without sapience**

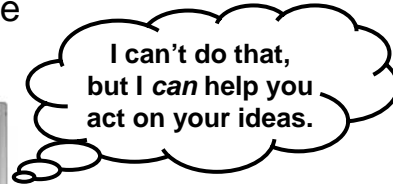
## Oh no! What Is *Sapience*?



- A sapient activity is one that requires a thinking human to perform
- A non-sapient activity can be performed by
  - a machine (quickly and precisely)
  - or by a human that has decided NOT to think (slowly and fallibly)
  - looks like machines win there, right?
- BUT our job is not merely to test for repeatability, but also for *adaptability and value*

# Testing IS Exploring

- Testing as I see it is all about exploration, discovery, investigation, and learning
  - Testing can be assisted by machines, but can't be done by machines alone
  - It is a *sapient* process



See <http://www.developsense.com/2009/08/testing-vs-checking.html>



# Automation Can't...



# Exploratory Testing



## The way we practice and teach it, exploratory testing...

- **IS NOT** “random testing” (or sloppy, or slapdash testing)
- **IS NOT** “unstructured testing”
- **IS NOT** procedurally structured
- **IS NOT** unteachable
- **IS NOT** unmanageable
- **IS NOT** scripted
- **IS NOT** a technique
- **IS** “ad hoc”, in the dictionary sense, “to the purpose”
- **IS** structured and rigorous
- **IS** cognitively structured
- **IS** highly teachable
- **IS** highly manageable
- **IS** chartered
- **IS** an approach

**What you do next is governed by  
what you're learning**

# Contrasting Approaches



## Scripted Testing

- Is directed from elsewhere
- Is determined in advance
- Is about confirmation
- Is about controlling tests
- Emphasizes predictability
- Emphasizes decidability
- Like making a speech
- Like playing from a score

## Exploratory Testing

- Is directed from within
- Is determined in the moment
- Is about investigation
- Is about improving test design
- Emphasizes adaptability
- Emphasizes learning
- Like having a conversation
- Like playing in a jam session

**The tester's mind is in control,  
not the script.**

## To Learn Excellent Exploratory Testing We Must Learn To *Test*

- Learning how to test in an exploratory way can be challenging, because:

**Nobody ever taught us how to test.**

- **WHEREAS...**

- Almost nobody enjoys reviewing written test procedures.
- Almost nobody knows how to evaluate the quality of written test procedures.
- Almost every manager seems to think that written tests are Good Things.

- **THEREFORE**

- Writing *awful* test procedures won't get us fired. Some companies will even *reward us* for the poor quality of our test procedures.

- **and**

- That means there is little pressure on us to become excellent testers.

# Exploratory Testing IS Structured



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

Not *procedurally* structured, but *cognitively* structured.

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


# Oracles



An *oracle* is  
a heuristic principle  
or mechanism  
by which  
someone  
might recognize a problem.

(usually works, might fail)

(but not decide conclusively)

 **Bug (n): Something that bugs someone who matters**

## All Test Oracles Are Heuristic



- Oracles (and heuristics) are fallible and context-dependent.
- Oracles can be contradicted by other oracles.
- Multiple oracles may increase our confidence, but even combinations of oracles are fallible.
- There is no single oracle that can tell us whether a program (or feature) is working correctly at all times and in all circumstances.
- Recognizing a different problem usually requires a different oracle.
- A tester doesn't need to be aware of an oracle in advance of the observation, *unless the test is designed to be run by rote*—that is, unless it's a *check*.
- Any time you see a problem, you *must* be using an oracle... so *what is it?*

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



**History**  
**Image**  
**Comparable Products**  
**Claims**  
**User Expectations**  
**Purpose**  
**Product**  
**Standards**

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



# Coverage Isn't Just Code Coverage

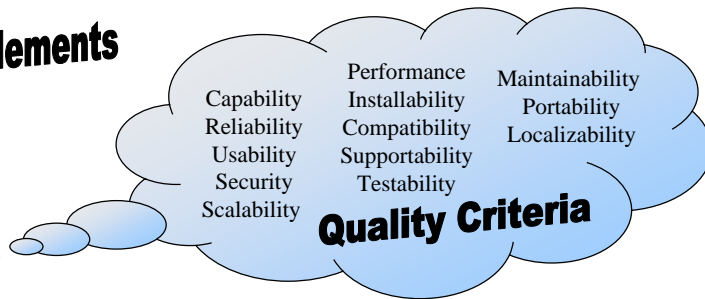


*Test coverage is the amount of the system space that has been tested.*

**There are as many kinds of coverage as there are ways to model the product.**

## **Product Elements**

- Structure
- Functional
- Data
- Platform
- Operations
- Time



## Cost as a Simplifying Factor

### *Try quick tests as well as careful tests*



A *quick test* is a cheap test that has some value but requires little preparation, knowledge, or time to perform.

- Happy Path
- Tour the Product
  - *Sample Data*
  - *Variables*
  - *Files*
  - *Complexity*
  - *Menus & Windows*
  - *Keyboard & Mouse*
- Interruptions
- Undermining
- Adjustments
- Dog Piling
- Continuous Use
- Feature Interactions
- Click on Help

18

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

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

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

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

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

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

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

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

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

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

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

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

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

## Cost as a Simplifying Factor

### *Try quick tests as well as careful tests*

A *quick test* is a cheap test that has some value but requires little preparation, knowledge, or time to perform.

- Input Constraint Attack
- Click Frenzy
- Shoe Test
- Blink Test
- Error Message Hangover
- Resource Starvation
- Multiple Instances
- Crazy Configs
- Cheap Tools

19

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

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

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

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

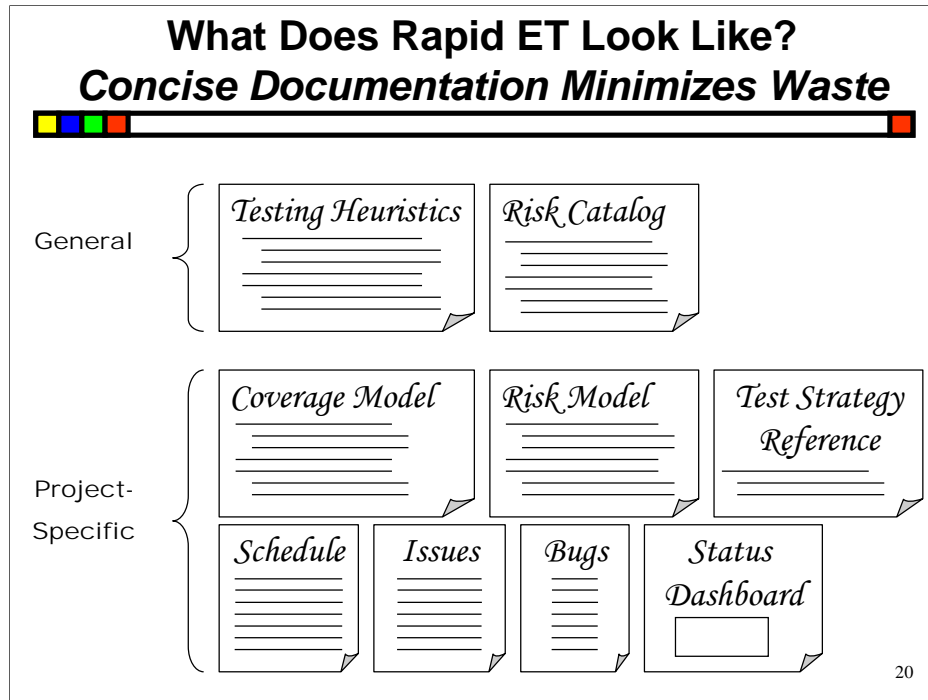
**Error Message Hangover:** Make error messages happen and test hard after they are dismissed. Often developers handle errors poorly.

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

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

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

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

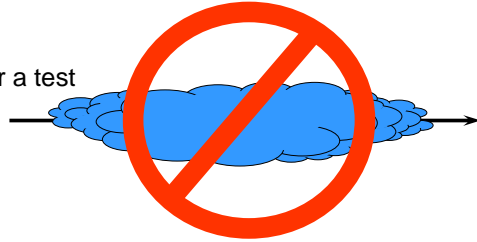


Detailed procedural documentation is expensive and largely unnecessary.

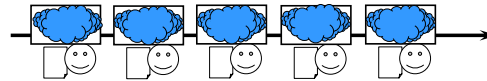
Tutorial documentation is also usually unnecessary, but if you do it, then keep it separate from the working documents.

## Accountability for Exploratory Testing: Session-Based Test Management

- Charter
  - A clear, concise mission for a test session
- Time Box
  - 90-minutes (+/- 45)
- Reviewable Results
  - a session sheet—a test report whose raw data can be scanned, parsed and compiled by a tool
- Debriefing
  - a conversation between tester and manager or test lead



**VS.**



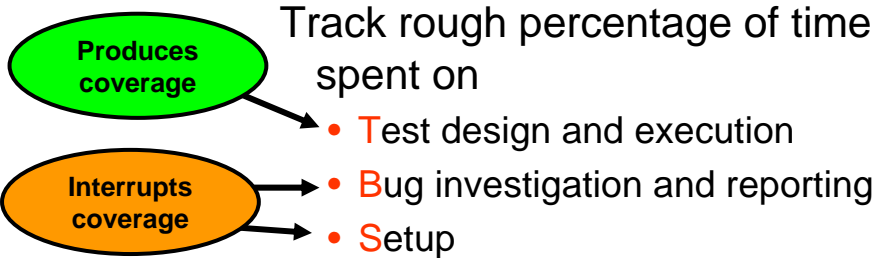
For more info, see <http://www.satisfice.com/sbtm>

## Charter: A Clear Mission for the Session



- From one to three sentences
- May suggest what should be tested, how it should be tested, and what problems to look for
- May refer to other documents or information sources
- *A charter is not meant to be a detailed plan.*
- General charters may be necessary at first:
  - “Analyze the Insert Picture function. Create a test coverage outline and risk list to guide future sessions.”
- Specific charters provide better focus, but take more effort to design:
  - “Test clip art insertion. Focus on stress and flow techniques, and make sure to insert into a variety of documents. We’re concerned about resource leaks or anything else that might degrade performance over time.”

# How To Measure ET Effectiveness



Ask why time was spent on each:

- Lots on T *might* indicate great code, but *might* indicate poor bug-finding skill
- Lots on B *might* mean code quality problems, but *might* suggest inefficiency in reporting
- Lots on S *might* mean testability or configuration problems for customers, or it *might* mean early days of testing

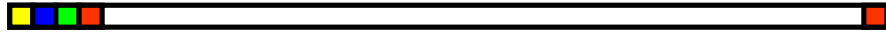
# How To Measure Test Coverage



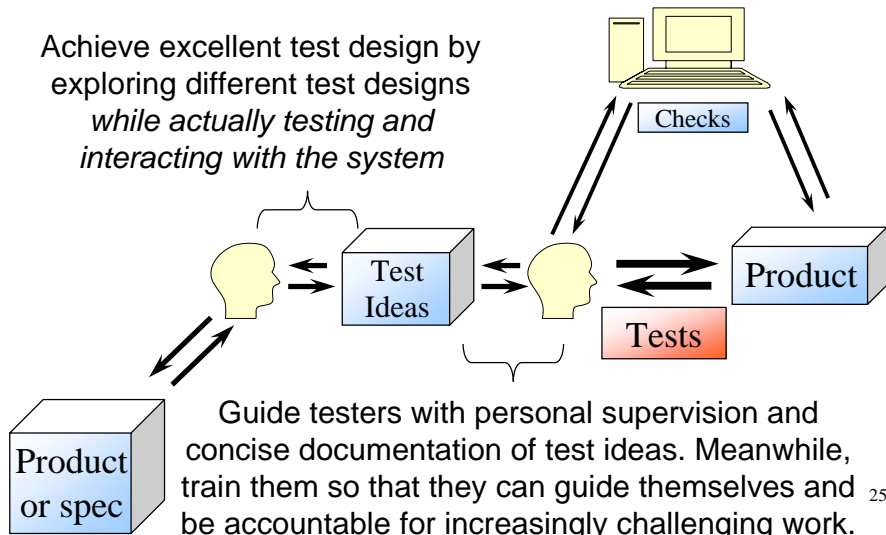
- *Test* coverage isn't merely *code* coverage
- Identify quality criteria, and identify session time *focused* on each criterion
- Consider product elements (structure, function, data, platform, operations, and time); break them down into coverage areas
- Assess test coverage in terms of
  - Level 1: Smoke and sanity
  - Level 2: Common, core, critical aspects
  - Level 3: Complex, challenging, harsh, extreme, exceptional



# How To Manage Exploratory Testing



Achieve excellent test design by exploring different test designs *while actually testing and interacting with the system*



# Acknowledgements



- James Bach (<http://satisfice.com>)
- Cem Kaner (<http://www.kaner.com>)
- Thanks to Chad Wathington for his collaboration on this talk

Questions? More information?

**Michael Bolton**

<http://www.developsense.com>

[michael@developsense.com](mailto:michael@developsense.com)

# Readings



- *Perfect Software and Other Illusions About Testing*
- *Quality Software Management, Vol. 1: Systems Thinking*
- *Quality Software Management, Vol. 2: First Order Measurement*
- *Exploring Requirements: Quality Before Design*
  - Gerald M. Weinberg
- *Lessons Learned in Software Testing*
  - Kaner, Bach, and Pettichord
- DevelopSense Web Site (and blog), <http://www.developsense.com>
  - Michael Bolton
- Satisfice Web Site (and blog), <http://www.satisfice.com>
  - James Bach
- Collaborative Software Testing, <http://www.kohl.ca>
  - Jonathan Kohl
- Quality Tree Software, <http://www.qualitytree.com>
  - Elisabeth Hendrickson