

Black Box Software Testing

(Professional Seminar)

Cem Kaner, J.D., Ph.D.

Professor of Computer Sciences
Florida Institute of Technology

Section:17

Exploratory Testing

Summer, 2002

Contact Information:

kaner@kaner.com

www.kaner.com (testing website)

www.badsoftware.com (legal website)

I grant permission to make digital or hard copies of this work for personal or classroom use, with or without fee, provided that (a) copies are not made or distributed for profit or commercial advantage, (b) copies bear this notice and full citation on the first page, and if you distribute the work in portions, the notice and citation must appear on the first page of each portion, (c) each page bear the notice "Copyright (c) Cem Kaner" or if you changed the page, "Adapted from Notes Provided by Cem Kaner". Abstracting with credit is permitted. The proper citation for this work is Cem Kaner, *A Course in Black Box Software Testing (Professional Version)*, Summer-2002, www.testing-education.org. To copy otherwise, to republish or post on servers, or to distribute to lists requires prior specific permission and a fee. Request permission to republish from kaner@kaner.com.

Black Box Software Testing

Paradigms: Exploratory Testing

Several of these slides are from James Bach, with permission, or from materials co-authored with James Bach

Acknowledgements

Many of the ideas in these notes were reviewed and extended by my colleagues at the 7th Los Altos Workshop on Software Testing. I appreciate the assistance of the other LAWST 7 attendees: Brian Lawrence, III, Jack Falk, Drew Pritsker, Jim Bampos, Bob Johnson, Doug Hoffman, Chris Agruss, Dave Gelperin, Melora Svoboda, Jeff Payne, James Tierney, Hung Nguyen, Harry Robinson, Elisabeth Hendrickson, Noel Nyman, Bret Pettichord, & Rodney Wilson.

Exploratory Testing

Simultaneously:

- Learn about the product
- Learn about the market
- Learn about the ways the product could fail
- Learn about the weaknesses of the product
- Learn about how to test the product
- Test the product
- Report the problems
- Advocate for repairs
- *Develop new tests based on what you have learned so far.*

Exploratory Testing

Tag line

- “Simultaneous learning, planning, and testing.”

Fundamental question or goal

- Software comes to tester under-documented and/or late. Tester must simultaneously learn about the product and about the test cases / strategies that will reveal the product and its defects.

Paradigmatic case(s)

- Skilled exploratory testing of the full product
- Rapid testing
- Emergency testing (including thrown-over-the-wall test-it-today testing.)
- Third party components.
- Troubleshooting / follow-up testing of defects.

Exploratory Testing

Strengths

- Customer-focused, risk-focused
- Takes advantage of each tester's strengths
- Responsive to changing circumstances
- Well managed, it avoids duplicative analysis and testing
- High bug find rates

Blind spots

- The less we know, the more we risk missing.
- Limited by each tester's weaknesses (can mitigate this with careful management)
- This is skilled work, juniors aren't very good at it.

Regression vs Exploration

Regression

- Inputs:
 - » old test cases and analyses leading to new test cases
- Outputs:
 - » archival test cases, preferably well documented, and bug reports
- Better for:
 - » reuse across multi-version products

Exploration

- Inputs:
 - » models or other analyses that yield new tests
- Outputs
 - » scribbles and bug reports
- Better for:
 - » Find new bugs, scout new areas, risks, or ideas

Doing Exploratory Testing

Keep your mission clearly in mind.

Distinguish between testing and observation.

While testing, be aware of the limits of your ability to detect problems.

Keep notes that help you report what you did, why you did it, and support your assessment of product quality.

Keep track of questions and issues raised in your exploration.

Problems to be Wary of...

Habituation may cause you to miss problems.

Lack of information may impair exploration.

Expensive or difficult product setup may increase the cost of exploring.

Exploratory feedback loop may be too slow.

Old problems may pop up again and again.

High MTBF may not be achievable without well defined test cases and procedures, in addition to exploratory approach.

Styles of Exploration

Experienced, skilled explorers develop their own styles.

When you watch or read different skilled explorers, you see very different approaches. This is a survey of the approaches that I've seen.

Characterizing the Styles

At the heart of all of the approaches to exploratory testing (all of the styles), I think we find *questions* and *questioning skills*. As you consider the styles that follow, try to characterize them with respect to each other:

- All of the approaches are methodical, but do they focus on the
 - » Method of questioning?
 - » Method of describing or analyzing the product?
 - » The details of the product?
 - » The patterns of use of the product?
 - » The environment in which the product is run?
- To what extent would this style benefit from group interaction?

Characterizing the Styles

- What skills and knowledge does the style require or assume?
 - » Programming / debugging
 - » Knowledge of applications of this type and how they fail
 - » Knowledge of the use of applications of this type
 - » Deep knowledge of the software under test
 - » Knowledge of the system components (h/w or s/w or network) that are the context for the application
 - » Long experience with software development projects and their typical problems
 - » Requirements analysis or problem decomposition techniques
 - » Mathematics, probability, formal modeling techniques

Query: Are any of these techniques appropriate to novices? Can we train novices in exploration?

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Styles of Exploration

- Basics
 - » **“Random”**
 - » **Questioning**
 - » **Similarity to previous errors**
 - » **Following up gossip and predictions**
 - » **Follow up recent changes**
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Random

- People who don't understand exploratory testing describe it as “random testing.” They use phrases like “random tests”, “monkey tests”, “dumb user tests”. This is probably the most common characterization of exploratory testing.
- This describes very little of the type of testing actually done by skilled exploratory testers.

Questioning

Questioning is the essence of exploration. The tester who constantly asks good questions can

- Avoid blind spots
- Quickly think of new test cases
- Constantly vary our approaches and targets
- Discover holes in specifications and product descriptions

These notes collect my material on questioning into one section, presented later as part of test planning.

Similarity to Previous Errors

James Bach once described exploratory testers as

mental pack rats who horde memories of every bug they've ever seen.

The way they come up with cool new tests is by analogy:

Gee, I saw a program kind of like this before, and it had a bug like this.

How could I test this program to see if it has the same old bug?

A more formal variation:

- Create a potential bugs list, like the Appendix A of *Testing Computer Software*

Another related type of analogy:

- Sample from another product's test docs.

Follow Up Gossip And Predictions

Sources of gossip:

- directly from programmers, about their own progress or about the progress / pain of their colleagues
- from attending code reviews (for example, at some reviews, the question is specifically asked in each review meeting, “What do you think is the biggest risk in this code?”)
- from other testers, writers, marketers, etc.

Sources of predictions

- notes in specs, design documents, etc. that predict problems
- predictions based on the current programmer’s history of certain types of defects

Follow Up Recent Changes

Given a current change

- tests of the feature / change itself
- tests of features that interact with this one
- tests of data that are related to this feature or data set
- tests of scenarios that use this feature in complex ways

Styles of Exploration

- Hunches
- **Models**
 - » **Architecture diagrams**
 - » **Bubble diagrams**
 - » **Data relationships**
 - » **Procedural relationships**
 - » **Model-based testing (state matrix)**
 - » **Requirements definition**
 - » **Functional relationships (for regression testing)**
 - » **Failure models**
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Models and Exploration

We usually think of modeling in terms of preparation for formal testing, but there is no conflict between modeling and exploration. Both types of tests start from models. The difference is that in exploratory testing, our emphasis is on execution (try it *now*) and learning from the results of execution rather than on documentation and preparation for later execution.

Architecture Diagrams

Work from a high level design (map) of the system

- pay primary attention to interfaces between components or groups of components. We're looking for cracks that things might have slipped through
- what can we do to screw things up as we trace the flow of data or the progress of a task through the system?

You can build the map in an architectural walkthrough

- Invite several programmers and testers to a meeting. Present the programmers with use cases and have them draw a diagram showing the main components and the communication among them. For a while, the diagram will change significantly with each example. After a few hours, it will stabilize.
- Take a picture of the diagram, blow it up, laminate it, and you can use dry erase markers to sketch your current focus.
- Planning of testing from this diagram is often done jointly by several testers who understand different parts of the system.

Bubble (Reverse State) Diagrams

To troubleshoot a bug, a programmer will often work the code backwards, starting with the failure state and reading for the states that could have led to it (and the states that could have led to those).

The tester imagines a failure instead, and asks how to produce it.

- Imagine the program being in a failure state. Draw a bubble.
- What would have to have happened to get the program here? Draw a bubble for each immediate precursor and connect the bubbles to the target state.
- For each precursor bubble, what would have happened to get the program there? Draw more bubbles.
- More bubbles, etc.
- Now trace through the paths and see what you can do to force the program down one of them.

Bubble (Reverse State) Diagrams

Example:

- *How could we produce a paper jam (as a result of defective firmware, rather than as a result of jamming the paper?) The laser printer feeds a page of paper at a steady pace. Suppose that after feeding, the system reads a sensor to see if there is anything left in the paper path. A failure would result if something was wrong with the hardware or software controlling or interpreting the paper feeding (rollers, choice of paper origin, paper tray), paper size, clock, or sensor.*

Data Relationships

- Pick a data item
- Trace its flow through the system
- What other data items does it interact with?
- What functions use it?
- Look for inconvenient values for other data items or for the functions, look for ways to interfere with the function using this data item

Data Relationship Chart

<i>Field</i>	<i>Entry Source</i>	<i>Display</i>	<i>Print</i>	<i>Related Variable</i>	<i>Relationship</i>
Variable 1	<i>Any way you can change values in V1</i>	<i>After V1 & V2 are brought to incompatible values, what are all the ways to display them?</i>	<i>After V1 & V2 are brought to incompatible values, what are all the ways to display or use them?</i>	Variable 2	Constraint to a range
Variable 2	<i>Any way you can change values in V1</i>			Variable 1	Constraint to a range

Procedural Relationships

- Pick a task
- Step by step, describe how it is done and how it is handled in the system (to as much detail as you know)
- Now look for ways to interfere with it, look for data values that will push it toward other paths, look for other tasks that will compete with this one, etc.

Improvisational Testing

The originating model here is of the test effort, not (explicitly) of the software.

Another approach to ad hoc testing is to treat it as improvisation on a theme, not unlike jazz improvisation in the musical world. For example, testers often start with a Test Design that systematically walks through all the cases to be covered. Similarly, jazz musicians often start with a musical score or “lead sheet” for the tunes on which they intend to improvise.

In this version of the ad hoc approach, the tester is encouraged to take off on tangents from the original Test Design whenever it seems worthwhile. In other words, the tester uses the test design but invents variations. This approach combines the strengths of both structured and unstructured testing: the feature is tested as specified in the test design, but several variations and tangents are also tested. On this basis, we expect that the improvisational approach will yield improved coverage.

Improvisational Testing

The originating model here is of the test effort, not (explicitly) of the software.

Improvisational techniques are also useful when verifying that defects have been fixed. Rather than simply verifying that the steps to reproduce the defect no longer result in the error, the improvisational tester can test more deeply “around” the fix, ensuring that the fix is robust in a more general sense.

**Johnson & Agruss, Ad Hoc Software Testing:
Exploring the Controversy of Unstructured Testing
STAR'98 WEST**

State Model-Based Testing

Notes from Harry Robinson & James Tierney

By modeling specifications, drawing finite state diagrams of what we thought was important about the specs, or just looking at the application or the API, we can find orders of magnitude more bugs than traditional tests.

Example, they spent 5 hours looking at the API list, found 3-4 bugs, then spent 2 days making a model and found 272 bugs. The point is that you can make a model that is too big to carry in your head. Modeling shows inconsistencies and illogicalities.

Look at

- all the possible inputs the software can receive, then
- all the operational modes, (something in the software that makes it work differently if you apply the same input)
- all the actions that the software can take.
- Do the cross product of those to create state diagrams so that you can see and look at the whole model.
- Use to do this with dozens and hundreds of states, Harry has a technique to do thousands of states.

www.geocities.com/model_based_testing

Using a Model of the Requirements to Drive Test Design

Notes from Melora Svoboda

Requirements model based on Gause / Weinberg. Developing a mind map of requirements, you can find missing requirements before you see code.

Business requirements

- Issues
- Assumptions
- Choices
- <<<< the actual problem >>>>

Customer Problem Definition

- USERS (nouns)
 - *favored*
 - *disfavored*
 - *ignored*
- ATTRIBUTES (adjectives)
 - » defining
 - » optimizing

Using a Model of the Requirements to Drive Test Design

Notes from Melora Svoboda

Customer Problem Definition (continued)

- FUNCTIONS (verbs)
 - » hidden
 - » evident

The goal is to test the assumptions around this stuff, and discover an inventory of hidden functions.

Comment: This looks to me (Kaner) like another strategy for developing a relatively standard series of questions that fall out of a small group of categories of analysis, much like the Satisfice model. Not everyone finds the Satisfice model intuitive. If you don't, this might be a usefully different starting point.

Functional Relationships

(More notes from Melora)

A model (what you can do to establish a strategy) for deciding how to decide what to regression test after a change:

1. Map program structure to functions.
 - This is (or would be most efficiently done as) a glass box task. Learn the internal structure of the program well enough to understand where each function (or source of functionality) fits.
2. Map functions to behavioral areas (expected behaviors)
 - The program misbehaved and a function or functions were changed. What other behaviors (visible actions or options of the program) are influenced by the functions that were changed?
3. Map impact of behaviors on the data
 - When a given program behavior is changed, how does the change influence visible data, calculations, contents of data files, program options, or anything else that is seen, heard, sent, or stored?

Failure Model: Whittaker: “The fundamental cause of software errors”

Constraint violations

- input constraints
 - » such as buffer overflows
- output constraints
- computation
 - » look for divide by zeros and rounding errors. Figure out inputs that you give the system that will make it not recognize the wrong outputs.
- data violations
- Really good for finding security holes

Styles of Exploration

- Hunches
- Models
- **Examples**
 - » **Use cases**
 - » **Simple walkthroughs**
 - » **Positive testing**
 - » **Scenarios**
 - » **Soap operas**
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Use Cases

- List the users of the system
- For each user, think through the tasks they want to do
- Create test cases to reflect their simple and complex uses of the system

Simple Walkthroughs

Test the program broadly, but not deeply.

- Walk through the program, step by step, feature by feature.
- Look at what's there.
- Feed the program simple, nonthreatening inputs.
- Watch the flow of control, the displays, etc.

Positive Testing

- Try to get the program working in the way that the programmers intended it.
- One of the points of this testing is that you educate yourself about the program. You are looking at it and learning about it from a sympathetic viewpoint, using it in a way that will show you what the value of the program is.
- This is true “positive” testing—you are trying to make the program show itself off, not just trying to confirm that all the features and functions are there and kind of sort of working.

Scenarios

The ideal scenario has several characteristics:

- It is realistic (e.g. it comes from actual customer or competitor situations).
- There is no ambiguity about whether a test passed or failed.
- The test is complex, that is, it uses several features and functions.
- There is a stakeholder who will make a fuss if the program doesn't pass this scenario.

For more on scenarios, see the scenarios paradigm discussion.

Styles of Exploration

- Hunches
- Models
- Examples
- **Invariances**
- Interference
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Invariances

These are tests run by making changes that shouldn't affect the program. Examples:

- load fonts into a printer in different orders
- set up a page by sending text to the printer and then the drawn objects or by sending the drawn objects and then the text
- use a large file, in a program that should be able to handle any size input file (and see if the program processes it in the same way)
- mathematical operations in different but equivalent orders

=====

John Musa — Intro to his book, *Reliable Software Engineering*, says that you should use different values within an equivalence class. For example, if you are testing a flight reservation system for two US cities, vary the cities. They shouldn't matter, but sometimes they do.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- **Interference**
 - » **Interrupt**
 - » **Change**
 - » **Stop**
 - » **Pause**
 - » **Swap**
 - » **Compete**
- Error Handling
- Troubleshooting
- Group Insights
- Specifications

Interference Testing

We're often looking at asynchronous events here. One task is underway, and we do something to interfere with it.

In many cases, the critical event is extremely time sensitive. For example:

- An event reaches a process just as, just before, or just after it is timing out or just as (before / during / after) another process that communicates with it will time out listening to this process for a response. (“Just as?”—if special code is executed in order to accomplish the handling of the timeout, “just as” means during execution of that code)
- An event reaches a process just as, just before, or just after it is servicing some other event.
- An event reaches a process just as, just before, or just after a resource needed to accomplish servicing the event becomes available or unavailable.

Interrupt

Generate interrupts

- from a device related to the task (e.g. pull out a paper tray, perhaps one that isn't in use while the printer is printing)
- from a device unrelated to the task (e.g. move the mouse and click while the printer is printing)
- from a software event

Change

Change something that this task depends on

- swap out a floppy
- change the contents of a file that this program is reading
- change the printer that the program will print to (without signaling a new driver)
- change the video resolution

Stop

- Cancel the task (at different points during its completion)
- Cancel some other task while this task is running
 - » a task that is in communication with this task (the core task being studied)
 - » a task that will eventually have to complete as a prerequisite to completion of this task
 - » a task that is totally unrelated to this task

Pause

- Find some way to create a temporary interruption in the task.
- Pause the task
 - » for a short time
 - » for a long time (long enough for a timeout, if one will arise)
- Put the printer on local
- Put a database under use by a competing program, lock a record so that it can't be accessed — yet.

Swap (out of memory)

- Swap the process out of memory while it is running (e.g. change focus to another application and keep loading or adding applications until the application under test is paged to disk.
 - » Leave it swapped out for 10 minutes or whatever the timeout period is. Does it come back? What is its state? What is the state of processes that are supposed to interact with it?
 - » Leave it swapped out *much* longer than the timeout period. Can you get it to the point where it is supposed to time out, then send a message that is supposed to be received by the swapped-out process, then time out on the time allocated for the message? What are the resulting state of this process and the one(s) that tried to communicate with it?
- Swap a related process out of memory while the process under test is running.

Compete

Examples:

Compete for a device (such as a printer)

- put device in use, then try to use it from software under test
- start using device, then use it from other software
- If there is a priority system for device access, use software that has higher, same and lower priority access to the device before and during attempted use by software under test

Compete for processor attention

- some other process generates an interrupt (e.g. ring into the modem, or a time-alarm in your contact manager)
- try to do something during heavy disk access by another process

Send this process another job while one is underway

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- **Error Handling**
- Troubleshooting
- Group Insights
- Specifications

Error Handling

The usual suspects:

- Walk through the error list.
 - » Press the wrong keys at the error dialog.
 - » Make the error several times in a row (do the equivalent kind of probing to defect follow-up testing).
- Device-related errors (like disk full, printer not ready, etc.)
- Data-input errors (corrupt file, missing data, wrong data)
- Stress / volume (huge files, too many files, tasks, devices, fields, records, etc.)

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- **Troubleshooting**
- Group Insights
- Specifications

Troubleshooting

We often do exploratory tests when we troubleshoot bugs:

- Bug analysis:
 - » simplify the bug by deleting or simplifying steps
 - » simplify the bug by simplifying the configuration (or the tools in the background)
 - » clarify the bug by running variations to see what the problem is
 - » clarify the bug by identifying the version that it entered the product
 - » strengthen the bug with follow-up tests (using repetition, related tests, related data, etc.) to see if the bug left a side effect
 - » strengthen the bug with tests under a harsher configuration
- Bug regression: vary the steps in the bug report when checking if the bug was fixed

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- **Group Insights**
 - » **Brainstormed test lists**
 - » **Group discussion of related components**
 - » **Fishbone analysis**
- Specifications

Brainstormed Test Lists

We saw a simple example of this at the start of the class. You brainstormed a list of tests for the two-variable, two-digit problem:

- The group listed a series of cases (test case, why)
- You then examined each case and the class of tests it belonged to, looking for a more powerful variation of the same test.
- You then ran these tests.

You can apply this approach productively to any part of the system.

Group Discussion of Related Components

The objective is to test the interaction of two or more parts of the system.

The people in the group are very familiar with one or more of parts. Often, no one person is familiar with all of the parts of interest, but collectively the ideal group knows all of them.

The group looks for data values, timing issues, sequence issues, competing tasks, etc. that might screw up the orderly interaction of the components under study.

Fishbone Analysis

- Fishbone analysis is a traditional failure analysis technique. Given that the system has shown a specific failure, you work backwards through precursor states (the various paths that could conceivably lead to this observed failure state).
- As you walk through, you say that Event A couldn't have happened unless Event B or Event C happened. And B couldn't have happened unless B1 or B2 happened. And B1 couldn't have happened unless X happened, etc.
- While you draw the chart, you look for ways to prove that X (whatever, a precursor state) *could* actually have been reached. If you succeed, you have found one path to the observed failure.
- As an exploratory test tool, you use “risks” instead of failures. You imagine a possible failure, then walk backwards asking if there is a way to achieve it. You do this as a group, often with a computer active so that you can try to get to the states as you go.

Paired Exploratory Testing--Acknowledgment

The following, paired testing, slides developed out of several projects. We particularly acknowledge the help and data from participants in the First and Second Workshops on Heuristic and Exploratory Techniques (Front Royal, VA, November 2000 and March 2001, hosted by James Bach and facilitated by Cem Kaner), those being Jon Bach, Stephen Bell, Rex Black, Robyn Brilliant, Scott Chase, Sam Guckenheimer, Elisabeth Hendrickson, Alan A. Jorgensen, Brian Lawrence, Brian Marick, Mike Marduke, Brian McGrath, Erik Petersen, Brett Pettichord, Shari Lawrence Pfleeger, Becky Winant, and Ron Wilson.

Additionally, we thank Noel Nyman and Ross Collard for insights and James Whittaker for co-hosting one of the two paired testing trials at Florida Tech.

A testing pattern on paired testing was drafted by Brian Marick, based on discussions at the Workshop on Patterns of Software Testing (POST 1) in Boston, January 2001 (hosted primarily by Sam Guckenheimer / Rational and Brian Marick, facilitated by Marick). The latest draft is at "Pair Testing" pattern) (<<http://www.testing.com/test-patterns/patterns/pair-testing.pdf>>).

Paired Exploratory Testing

- **Based on our (and others') observations of effective testing workgroups at several companies. We noticed several instances of high productivity, high creativity work that involved testers grouping together to analyze a product or to scheme through a test or to run a series of tests. We also saw/used it as an effective training technique.**
- **In 2000, we started trying this out, at WHET, at Satisfice, and at one of Satisfice's clients. The results were spectacular. We obtained impressive results in quick runs at Florida Tech as well, and have since received good reports from several other testers.**

Paired Programming

- Developed independently of paired testing, but many of the same problems and benefits apply.
- The eXtreme Programming community has a great deal of experience with paired work, much more than we do, offers many lessons:
 - » Kent Beck, *Extreme Programming Explained*
 - » Ron Jeffries, Ann Anderson & Chet Hendrickson, *Extreme Programming Installed*
- Laurie Williams of NCSU does research in pair programming. For her publications, see <http://collaboration.csc.ncsu.edu/laurie/>

What is Paired Testing

- Two testers and (typically) one machine.
- Typically (as in XP)
 - » Pairs work together voluntarily. One person might pair with several others during a day.
 - » A given testing task is the responsibility of one person, who recruits one or more partners (one at a time) to help out.
- We've seen stable pairs who've worked together for years.
- One tester strokes the keys (but the keyboard may pass back and forth in a session) while the other suggests ideas or tests, pays attention and takes notes, listens, asks questions, grabs reference material, etc.

A Paired Testing Session

Start with a charter

- Testers might operate from a detailed project outline, pick a task that will take a day or less
- Might (instead or also) create a flipchart page that outlines this session's work or the work for the next few sessions.
 - » An exploratory testing session lasts about 60-90 minutes.
- The charter for a session might include what to test, what tools to use, what testing tactics to use, what risks are involved, what bugs to look for, what documents to examine, what outputs are desired, etc.

Benefits of Paired Testing

- Pair testing is different from many other kinds of pair work because testing is an **idea generation activity** rather than a plan implementation activity. Testing is a heuristic search of an open-ended and multi-dimensional space.
- Pairing has the effect of forcing each tester to explain ideas and react to ideas. When one tester must phrase his thoughts to another tester, that simple process of phrasing seems to bring the ideas into better focus and naturally triggers more ideas.
- If faithfully performed, we believe this will result in more and better ideas that inform the tests.

Benefits of Paired Testing

Generate more ideas

- Naturally encourages creativity
- More information and insight available to apply to analysis of a design or to any aspect of the testing problem
- Supports the ability of one tester to stay focused and keep testing. This has a major impact on creativity.

More fun

Benefits of Paired Testing

Helps the tester stay on task. Especially helps the tester pursue a streak of insight (an exploratory vector).

- A flash of insight need not be interrupted by breaks for note-taking, bug reporting, and follow-up replicating. The non-keyboard tester can:
 - » Keep key notes while the other follows the train of thought
 - » Try to replicate something on a second machine
 - » Grab a manual, other documentation, a tool, make a phone call, grab a programmer--get support material that the other tester needs.
 - » Record interesting candidates for digression

Also, the fact that two are working together limits the willingness of others to interrupt them, especially with administrivia.

Benefits of Paired Testing

Better Bug Reporting

- Better reproducibility
- Everything reported is reviewed by a second person.
- Sanity/reasonability check for every design issue
 - » (example from Kaner/Black on Star Office tests)

Great training

- Good training for novices
- Keep learning by testing with others
- Useful for experienced testers when they are in a new domain

Benefits of Paired Testing

Additional technical benefits

- Concurrency testing is facilitated by pairs working with two (or more) machines.
- Manual load testing is easier with several people.
- When there is a difficult technical issue with part of the project, bring in a more knowledgeable person as a pair

Risks and Suggestions

- Paired testing is *not* a vehicle for fobbing off errand-running on a junior tester. The pairs are partners, the junior tester is often the one at the keyboard, and she is always allowed to try out her own ideas.
- Accountability must belong to one person. Beck and Jeffries, et al. discuss this in useful detail. One member of the pair owns the responsibility for getting the task done.
- Some people are introverts. They need time to work alone and recharge themselves for group interaction.
- Some people have strong opinions and don't work well with others. Coaching may be essential.

Risks and Suggestions

Have a coach available.

- Generally helpful for training in paired testing and in the conduct of any type of testing
- When there are strong personalities at work, a coach can help them understand their shared and separate responsibilities and how to work effectively together.

Styles of Exploration

- Hunches
- Models
- Examples
- Invariances
- Interference
- Error Handling
- Troubleshooting
- Group Insight
- **Specifications**
 - » **Active reading -- Tripos**
 - » **Active reading -- Ambiguity analysis**
 - » **User manual**
 - » **Consistency heuristics**

Active Reading

- We cover James Bach's satisfice testing model in detail in another section (the section on questioning).
- You can use this method to discover faults in a specification, such as holes, ambiguities, and contradictions.
- The goal is to constantly question the spec, identifying statements about product, project and risk, but also identifying missing details and unrealistic discussions.
- Anything you flag as an issue (or write a question about), is a candidate for exploratory testing.

Active Reading

(Ambiguity Analysis)

There are all sorts of sources of ambiguity in software design and development.

- In the wording or interpretation of specifications or standards
- In the expected response of the program to invalid or unusual input
- In the behavior of undocumented features
- In the conduct and standards of regulators / auditors
- In the customers' interpretation of their needs and the needs of the users they represent
- In the definitions of compatibility among 3rd party products

Whenever there is ambiguity, there is a strong opportunity for a defect (at least in the eyes of anyone who understands the world differently from the implementation).

One interesting workbook: Cecile Spector, *Saying One Thing, Meaning Another*.

User Manual

Write part of the user manual and check the program against it as you go. Any writer will discover bugs this way. An exploratory tester will discover quite a few this way.

Consistency Heuristics:

Consistent with History: Present function behavior is consistent with past behavior.

Consistent with an Image: Function behavior is consistent with an image that the organization wants to project.

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

Consistent with Claims: Function behavior is consistent with what people say it's supposed to be.

Consistent with User Values: Function behavior is consistent with what we think users want.

Consistent within Product: Function behavior is consistent with behavior of comparable functions or functional patterns within the product.

Consistent with Purpose: Function behavior is consistent with its apparent purpose.

Exploratory Testing: Some Papers of Interest

- Chris Agruss & Bob Johnson, *Ad Hoc Software Testing Exploring the Controversy of Unstructured Testing*
- Cem Kaner & James Bach, *Exploratory Testing.*
(Available later in the materials)
- Whittaker, *How to Break Software*

Sample Exam Questions

Imagine that you were testing the feature, “Enters and/or edits matrices.”

Describe four examples of each of the following types of attacks that you could make on this feature, and for each one, explain why your example is a good attack of that kind.

- Input constraint attacks
- Output constraint attacks
- Storage constraint attacks
- Computation constraint attacks.

(Notes for you while you study. Refer to Jorgensen / Whittaker’s paper on how to break software. Don’t give me two examples of what is essentially the same attack. In the exam, I will not ask for all 16 examples, but I might ask for 4 examples of one type or two examples of two types, etc.)

Sample Exam Questions

Imagine that you were testing the feature, “Graphs functions and plots statistical data.”

Describe four examples of each of the following types of attacks that you could make on this feature, and for each one, explain why your example is a good attack of that kind.

- Input constraint attacks
- Output constraint attacks
- Storage constraint attacks
- Computation constraint attacks.

(Notes for you while you study. Refer to Jorgensen / Whittaker’s paper on how to break software. Don’t give me two examples of what is essentially the same attack. In the exam, I will not ask for all 16 examples, but I might ask for 4 examples of one type or two examples of two types, etc.)

Sample Exam Questions

Imagine that you were testing the feature, “Transfers data to or from a connected calculator.”

Describe four examples of each of the following types of attacks that you could make on this feature, and for each one, explain why your example is a good attack of that kind.

- Input constraint attacks
- Output constraint attacks
- Storage constraint attacks
- Computation constraint attacks.

(Notes for you while you study. Refer to Jorgensen / Whittaker’s paper on how to break software. Don’t give me two examples of what is essentially the same attack. In the exam, I will not ask for all 16 examples, but I might ask for 4 examples of one type or two examples of two types, etc.)

