Styles of Exploration

Experienced, skilled explorers develop their own styles.
Here's a survey of some of the styles we've seen.

Cem Kaner & Bob Johnson

LAWST 7

The ideas in this presentation were reviewed and extended by our colleagues at the 7th Los Altos Workshop on Software Testing. We 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.

Styles

When you watch or read different skilled explorers, you see very different approaches. At the heart of all of the approaches, I think we find *questions* and *questioning skills*. (But then, I'm the guy who defines a test case as a unified or single question that you ask the program, so maybe I'm biased.) As you consider the styles that follow, think about themes.

- 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?
- 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?

How it’s done: An Illustration

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

So, how do you do it? This is a difficult process to describe, because its main virtues are a lack of rigid structure, and the freedom to try alternative pathways! Also, much of what experienced software testers do is highly intuitive, rather than strictly logical. However, here are a few tricks and techniques that will help you to do effective ad hoc testing.

To begin with, we suggest that you target areas that are not already covered very thoroughly by your test designs. In our experience, test designs are written to cover specific features in the software under test, with relatively little attention paid to the potential interaction between features. Much of this interaction goes on at the subsystem level, because it supports multiple features (e.g. the graphical subsystem). Imagine some potential interactions such as these that could go awry in your program, and then set out to test your theory using an ad hoc approach.

Before embarking on your ad hoc adventure, take out paper and pencil. On the paper, write down what you’re most interested in learning about the program during this session. Be specific. Note exactly what you plan to achieve during this testing, and how long you are going to spend doing the work. Then make a short list of what might go wrong and how you would be able to detect the problem.

Next, start the program, and try to exercise the features that you are currently concerned about. Remember that you want to use this time to better understand the program, so side trips to explore both breadth and depth of the program are very much in order.

As an example, suppose you want to determine how thoroughly you need to test a program’s garbage collection abilities. From the documentation given, you will probably be uncertain if any garbage collection is required or exists. Although many programs use some form of garbage collection, this will seldom be mentioned in any specification. Systematic testing of garbage collection is very time consuming. However, with one day’s ad hoc testing of most programs, an experienced tester can determine if the program’s garbage collection is “okay,” has “some problems,” or is “seriously broken.” With this information the Test Manager can determine how much effort to spend in testing this further. We’ve even seen the development manager withdraw features for review, and call for a rewrite of the code as a result of this knowledge, saving both testing and debugging time.

There are hundreds of these low-level housekeeping functions, often re-implemented for each new program, and prone to design and programmer error. Memory management, sorting, searching, two-phase commit, hashing, saving to disk (include temporary and cache file), menu navigation, and parsing are just a few examples.

A good ad hoc tester needs to understand the design goals and requirements for these low-level functions (see suggested reading, below). What choices did the development team make, and what were the weaknesses of those choices? Ad hoc testing can be done as black box testing. However, this means the tester must check for all the major design patterns that might have been used. Clear or white box testing allows the tester to narrow the testing to known problems.

Read defect reports from many projects, not just from your project. Your defect database doesn’t tell you what kind of mistakes the developers are making; it tells you what kinds of mistakes you are finding. You want to find new types of problems. Expand your horizon. Read about problems/defects/weaknesses in the application’s environment. Sources of such problems include the operating system, the language being used, the specific compiler, the libraries used, and the APIs being called.

Learn to use debuggers, profilers, and task monitors. In many cases you never see an error in the execution of the program; however, these tools can flag processes that are out of control. For this reason, among others, you need to use seasoned professionals to get the most out of your ad hoc testing.

Styles of Exploration

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

Styles of Exploration

- Hunches
  - **“Random”**
  - **“Perverse view”**
  - **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.

“Perverse” View

- The “perverse” tester reads (or hears) specs or other claims about how the program works and
  - intentionally misinterprets them,
  - intentionally does the task in an unusual (e.g. clumsy or indirect) order or
  - intentionally interferes with the program’s ability to fulfill them.

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 colleages
  - 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)**
- 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.
- **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 for a while, the system reads a sensor to see if there is anything left in the paper path. In this case, 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

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 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

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
  - 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 mind be a usefully different starting point.*

Data Relationship Chart

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

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?

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.

Scenarios

Some ways to trigger thinking about scenarios:

- Benefits-driven: People want to achieve X. How will they do it, for the following X's?
- Sequence-driven: People (or the system) typically does task X in an order. What are the most common orders (sequences) of subtasks in achieving X?
- Transaction-driven: We are trying to complete a specific transaction, such as opening a bank account or sending a message. What are all the steps, data items, outputs and displays, etc.?
- Get use ideas from competing product: Their docs, advertisements, help, etc., all suggest best or most interesting uses of their products. How would our product do these things?
- Competitor's output driven: Hey, look at these cool documents they can make. Look (think of Netscape's superb handling of often screwy HTML code) at how well they display things. How do we do with these?
- Customer's forms driven: Here are the forms the customer produces in her business. How can we work with (read, fill out, display, verify, whatever) them?

Soap Operas

- Build a scenario based on real-life experience. This means client/customer experience.
- Exaggerate each aspect of it:
  - example, for each variable, substitute a more extreme value
  - example, if a scenario can include a repeating element, repeat it lots of times
  - make the environment less hospitable to the case (increase or decrease memory, printer resolution, video resolution, etc.)
- Create a real-life story that combines all of the elements into a test case narrative.

(Thanks to Hans Buwalda for developing this approach and patiently explaining it to me.)

(As these have evolved, Hans distinguishes between *normal soap operas*, which combine many issues based on user requirements–typically derived from meetings with the user community and probably don’t exaggerate beyond normal use—and *killer soap operas*, which combine *and exaggerate to produce extreme cases*.)

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 equivalance 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.

Styles of Exploration

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

Active Reading

- **We covered James Bach's testing model in detail at the start of this section**
- **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.