Exploring Exploratory Testing

Andy Tinkham (FIT) andy@tinkham.org
Cem Kaner (FIT) kaner@kaner.com

The IEEE Computer Society[1] has been promoting a document called the *Software Engineering Body of Knowledge* (SWEBOK)[2]. This document is positioned to be a consensus document that focuses on generally accepted knowledge in software engineering.

According to SWEBOK (pages 5-9), exploratory testing is "the most widely practiced [testing] technique". "Tests are derived relying on tester skill and intuition…and on his/her experience with similar programs." SWEBOK advises "a more systematic approach" and says that exploratory testing "might be useful (but only if the tester is really expert!) to identify special tests not easily 'captured' by formalized techniques."

The SWEBOK's treatment of exploratory testing reflects the degree of controversy and confusion about exploratory testing in our field.

The fact is that every tester does exploratory testing. For example,

- When a tester finds a bug, she troubleshoots it a bit, both to find a simple set of reproduction conditions and to determine whether a variant of these conditions will yield a more serious failure. This is classic exploration. The tester decides what to do next based on what she's learned so far.
- When the programmer reports that the bug has been fixed, the tester runs the original failure-revealing test to see if the fix has taken. But the skilled tester also varies the regression test to see whether the fix was general enough and whether it had a side effect that broke some other part of the program. This is exploratory testing.
- When the tester first gets the product, with or without a specification, and tries out the features to see how they work, and then tries to do something "real" with the product to develop an appreciation of its design, this is exploratory testing.

Anyone who tests does exploratory testing, but some of us rely on exploration more heavily and more effectively than others.

This difference in opinion is the result of a prevailing belief that exploratory testing is primarily banging on the keyboard and avoiding the planning and work of "real" testing. Many testers have a very limited perception of what really is exploratory testing. In this paper, we hope to introduce the reader to a broader picture of what exploratory testing is and some of the components that make it a useful and powerful technique.

[1] But not the Association for Computing Machinery
[2] Guide to the Software Engineering Body of Knowledge, Trial Version 1.00, May 2001, http://www.swebok.org/documents/Trial_1_00/Trial_Version1_00_SWEBOK_Guide.pdf

This paper is an initial review within a larger project. Our goal is to develop an understanding of the typical tasks involved in exploratory testing and the skills and knowledge needed to perform these tasks. With that understanding, we can more effectively train people to be strong exploratory testers.

What is Exploratory Testing?

Definitions of exploratory testing have been offered by James Bach[3], Cem Kaner[4], Brian Marick,[5] Elisabeth Hendrickson,[6] and Chris Agruss and Bob Johnson.[7] All of these people have collaborated and polished each others' views. In 1999, these authors (and others) compared notes at the 7th Los Altos Workshop on Software Testing (LAWST VII) and highlighted some of the common characteristics of their views. These characteristics included:

- Interactive
- Concurrence of cognition and execution
- Creativity
- Drive towards fast results
- De-emphasize archived testing materials[8]

Bach also talks about tests spanning a continuum between purely scripted (in which the tester does precisely what the test script specifies and nothing else) to purely exploratory (in which the tester has no pre-planned activities to perform). Any given testing effort falls somewhere on this continuum. By performing this mapping, the focus of the effort becomes less "should we do exploratory testing?" than "to what degree is our testing exploratory?" Even predominantly pre-scripted testing can be exploratory when performed by a skilled tester. Any good tester will be alert to indications that something is wrong in the program. Should they see one of these indications, the tester will depart from the script they are executing and follow up on the problems hinted at by the indication they saw.

From this perspective, an appropriate definition of exploratory testing is "Any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests."[9]

We adopt that definition in this paper.

In the prototypic case (what Bach calls "freestyle exploratory testing"), exploratory testers continually learn about the software they're testing, the market for the product, the various ways in which the product could fail, the weaknesses of the product (including

[3] For example, "Exploratory Testing Explained" at http://www.satisfice.com/articles/et-article.pdf
[4] Black Box Testing course notes at http://www.testingeducation.org/
[5] http://www.testing.com/exploratory.html
[6] "Bug Hunting" course notes, received from the author
[7] http://www.testingcraft.com/ad_hoc_testing.pdf
[8] Pettichord, Bret, "An Exploratory Testing Workshop Report", http://www.testingcraft.com/exploratory-pettichord.html
[9] James Bach, "Exploratory Testing Explained", http://www.satisfice.com/articles/et-article.pdf

where problems have been found in the application historically and which developers tend to make which kinds of errors), and the best ways to test the software. At the same time that they're doing all this learning, exploratory testers also test the software, report the problems they find, advocate for the problems they found to be fixed, and develop new tests based on the information they've obtained so far in their learning.

What do Exploratory Testers Know?

Exploratory testers as a whole are difficult to stereotype. While all exploratory testing is based on knowledge (both the knowledge gained during the testing and the knowledge the exploratory tester brings to the testing process), the knowledge that the exploratory tester already has varies from person to person, just as with any sort of knowledge. For example, some testers have a great deal of familiarity with the application or type of application that they're testing. Other testers may be familiar with the platform that the application will run on or platforms similar to the one being used. Either of these testers (or yet other testers) may have knowledge of the risks associated with a particular product or platform type. Still others may have knowledge of specific test techniques or of tools and materials that are available to them.

Some testers base their tests on a sense of risk and their knowledge of how products can fail. This knowledge can come from many different places. Some of the knowledge these testers have is experiential – the tester worked on a project where a certain bug was found, and the tester remembers that bug and tests for it on future projects. The tester can also draw on the experience of other people (when they are aware of it), and need not have actually experienced the bug themselves. Bach has referred to the practitioners of this style of exploratory testing as "mental packrats who horde memories of every bug they've seen" and this image is quite fitting.

Exploratory testers may also draw on knowledge from training they've received (such as Elisabeth Hendrickson's techniques described in her Bug Hunting class) or from published sources, such as articles on ways an application (or applications) failed or on various testing techniques, or from published bug taxonomies (such as the one for online shopping cart applications, created by Giri Vijayaraghavan[10]) that catalog ways software can fail. James Whittaker[11] and Alan Jorgenson[12] have published work on attacks that can be used to try to find certain types of bugs in an application.

Bach has developed a model, called the Satisfice Heuristic Test Strategy Model[13], which shows the types of knowledge used by explorers. In this model, the project environment, the criteria defined for quality on the project, and elements of the product being tested, combine with test techniques to affect the perceived quality of the product. (Kaner also

[10] In his master's thesis, to be published at http://www.testingeducation.org. Also see "Bugs in Your Shopping Cart: A Taxonomy", by Giri Vijayaraghavan and Cem Kaner, at http://www.testingeducation.org/articles/ , winner of the Best Paper award at the 15th International Software Quality Conference

[11] *How to Break Software*, Whittaker, James; Addison-Wesley, 2003, ISBN: 0-201-79619-8

[12] "Software Design Based on Operational Modes" (available at http://www.cs.fit.edu/~tr/cs-2002-10.pdf) and "Testing with Hostile Data Streams" (available at http://www.cs.fit.edu/~tr/cs-2003-03.rtf)

[13] http://www.satisfice.com/tools/satisfice-tsm-4p.pdf

adds risk factors as a separate item that combines with the test techniques. Bach subsumes these into the quality criteria.) The model consists of several components for each of these elements and serves as a prompt that testers can use to determine the information they need for the project and then work with that information once they get it.

Each of these different areas of knowledge can be extremely helpful to an exploratory tester. Previous knowledge specific to the actual product being tested often takes the form of areas known to have been bug-ridden in the past or the tendency of the project team to make certain kinds of errors. This knowledge can then be used to make sure that the areas that historically have been problematic are given ample attention to ensure that any current problems in those areas are found. The same type of thing applies for all the other types of knowledge that a tester may have about a product, a platform or a combination of the two.

How else do Exploratory Testers Differ?

Kaner takes the approach of looking more at the profiles of individual testers who seem skilled at exploration and noticing that they show repeating patterns[14]. For example, a subject matter expert is more likely to come up with scenarios that rely heavily on knowledge of the product (focusing on product elements), while many testers master a small set of high-payoff techniques and look for elements of the product where they can apply them.

Other characteristics of exploratory testers also affect how they do their explorations. Everything from a tester's personality to her preferred learning styles to her past experiences has an impact on how the tester perceives risk, how they think that an application might fail, and how they design tests to find these failures and test for the risks they see. These all look like different "styles" of explorations, and we think they are separable in the sense, that, through training, you can (if you choose to) improve someone on one of them without much improving the others. For example, we can train someone to be a great interference tester or a great state model mapper, without having them learn the possible pitfalls that a particular type of software could fall into. But, ultimately, these styles are part of the broader toolkit and knowledge base shown in the Satisfice model, and testers who gain breadth with experience come to master many of these styles, instead of just one.

Questioning Strategies

Another way to approach the differences is to look at how various testers generate questions. In terms of practical applications, this is really just another test technique. However, we can use this technique to think about how we do testing instead of using the technique for actually doing the testing. For example, suppose we viewed each test case as a question we asked the product. These questions generally are of the form, "Can you [the product] do this?" or "What happens if I do that?" Each question correctly answered

[14] See Kaner's professional black box testing course notes available at http://www.testingeducation.org/ for more on the styles of exploratory testing and questioning strategies

increases our confidence in the application. If we consider testing the application this way, the problem of designing an appropriate set of test cases becomes a problem of designing a set of questions that help us get the best information we can about the product we're testing. Just like test cases, the set of possible questions we could ask is infinite for all practical purposes. We need to determine how we can generate a list of questions that will prove to be fruitful and manageable. If we can determine a great list of great questions, then finding tests to answer the questions might be straightforward.

Luckily for us, a great deal of research has been performed on how people generate great questions to solve problems. For example, the CIA has developed a set of questions called the Phoenix Checklist[15]. This set of questions is a list known as "context-free questions" and is designed "to encourage agents to look at a challenge from many different angles. Using Phoenix is like holding your challenge in your hand. You can turn it, look at it from underneath, see it from one view, hold it up to another position, imagine solutions, and really be in control of it."[16]

The Phoenix checklist begins by looking at the problem that we're trying to solve. It asks things like "Why is it necessary to solve the problem?" and "What isn't the problem?" to help the user get a clear grasp of what they're trying to accomplish and why. It then moves on to asking about similar problems (either that the problem solver has already solved, or problems which are similar in some aspect but easier to solve than the main problem being solved). From there, the checklist moves on to the plan for solving the problem, covering things like "What creative thinking techniques can you use to generate ideas?" and "How many different ways have you tried to solve the problem?" Finally, the checklist concludes with some questions that analyze the problem and solution to aid in future problem solving and determining whether the proposed solution is successful.

The Phoenix Checklist questions are called "context free questions" because they can be applied to any given context without changing the context of the problem they are being used to solve. The CIA is also not the only group to investigate this type of approach to problem solving. Donald Gauss and Gerald Weinberg look at the topic extensively in their book, *Exploring Requirements*, and a search on the web yields many more results, as well.

Another questioning strategy is the set of "newspaper questions" – the "who, what, where, when, how, and why" questions used by reporters to get as many of the details for their stories as they can. This list of questions is also geared to help the problem solver look at a problem from multiple angles and gain a better understanding of what they're trying to accomplish.

These strategies for investigating problems can be applied directly to exploratory testing. The checklists can be directly applied to the application and may yield insight directly. They also should be used to examine the testing process itself. Each testing organization has a mission, and this mission can vary from testing group to testing group. This mission

[15] Described in *ThinkerToys*, by Michael Michalko (pages 138-144)
[16] *ThinkerToys*, page 139

corresponds to the problem they're trying to solve. By applying context-free questions and the newspaper questions to achieving its mission, a testing group can make as efficient use of their time and resources as possible.

Ultimately, generating a list of questions is like generating a list of risks. You have to find a way to translate the question to the test. Exploratory testers do that in the moment, as they test.

The Role of Heuristics

Every decision that an exploratory tester makes is made under conditions of uncertainty and insufficient knowledge. Because of this, all decisions have some probability of being incorrect, and this probability means we can't mechanically choose the next thing to do. Instead, since each decision carries an element of risk, we turn to heuristics. Heuristics are a tool that people use to help them make decisions.

The *American Heritage Dictionary of the English Language* (3rd ed.) defines the term "heuristic" as "Of or relating to a usually speculative formulation serving as a guide in the investigation or solution of a problem." (It offers a second definition of "of, relating to, or constituting an educational method in which learning takes place through discoveries that result from investigations made by the student" which means that exploratory testing is a highly heuristic process that uses heuristics.) Examining the first definition shows us two interesting pieces. First, there's the phrase "a guide in the investigation or solution of a problem". Heuristics are guidelines that help us clarify or solve a problem (thus, in some ways, the Phoenix Checklist and newspaper questions are heuristics). The other interesting part of the definition is the phrase "usually speculative". Speculation is never guaranteed to be right – when someone speculates on something, she might be right or she might be far off base in her speculations. Heuristics are the same way. They are guidelines, but they are **only** guidelines. No heuristic is guaranteed to be applicable in every situation, and a heuristic that worked wonderfully in one situation may only make another situation worse. It is up to the heuristic user to evaluate the heuristics they consider using to determine whether a given heuristic is appropriate for his current situation. Even the process of evaluating a heuristic for its applicability can be an extremely useful process for helping an exploratory tester (or anyone else) to think about what he is trying to accomplish from different angles and perhaps help the person come up with a good solution to the problem at hand.

Bach has developed a wide range of heuristics and decision rules[17]. These heuristics either clearly describe what many of us believe testers actually do or seem to be compellingly useful suggestions for what testers *should* do. For example, in *Lessons Learned in Software Testing*, a list of heuristics for evaluating a test plan is presented. These heuristics can be used to consider how a test plan "fulfills its functions and how it meets the quality criteria." For example, one of the heuristics listed is "Important

[17] Notably in Chapter 11 of *Lessons Learned in Software Testing* (Kaner, Bach, and Pettichord, 2002), "General Functionality and Stability Test Procedure" (the Microsoft Windows 2000 Certification procedure, at http://www.satisfice.com/tools/procedure.pdf), and "Heuristics of Testability" (at http://www.satisfice.com/tools/testable.pdf)

problems fast". While it may usually be a good idea to organize a test effort so that the most important problems are found as quickly as possible to allow ample time for fixing them, there may be times when there's a need to test for less important problems first. Perhaps the marketing department wants screen shots to use in promoting the software, so the test group needs to find cosmetic and usability problems first to give the marketing department time to develop their materials with accurate screenshots, or maybe there is an upcoming demo for senior management, so the test group is tasked to find the things that senior management might pick up on and complain about, regardless of the importance of the bugs. Just like any heuristic, there are times when it's applicable and times when it isn't.

Other heuristics can guide a tester towards looking for specific kinds of failures. For example, the Consistency Heuristics, which Bach talks about in the "General Function and Stability Test Procedure" for Windows 2000 certification, illustrates ways that testers can determine whether their application behaves in a consistent manner. For example, one of the Consistency Heuristics is "consistent with comparable products". A tester using this heuristic to help guide his efforts would look at products similar to the one he was testing, and determine how the other products performed tasks. He would then see if the application he was testing did the same things in a consistent manner. This can be especially important when one of the comparable products is the market leader, and the tester's company is hoping to attract users who are already used to how the leading software does tasks.

Finally, heuristics can illustrate potential bugs directly. Bach's "mental packrats" who keep lists of every bug they've ever seen are using their lists of defects as heuristics. No application will have every bug on their list (hopefully!), but the lists serve as guides to the tester, helping her think of tests she might not have otherwise done.

For those of us who don't have the memory to keep long lists of defects (or even for those who do), risk taxonomies can serve as heuristics in guiding our tests. Taxonomies such as the one in *Testing Computer Software, 2nd ed.*[18] can serve as useful pointers to help an exploratory tester think of new ways to test his application. In the authors' lab at the Florida Institute of Technology, additional bug taxonomies are being completed. Giri Vijayaraghavan created the previously mentioned taxonomy for online shopping cart applications. Ajay Jha is currently working on developing a similar list of bugs for wireless handheld applications.

All of these taxonomies (and others) can be used to help a tester come up with more tests for her own application than she might otherwise have done. To use one of these taxonomies, a tester picks a risk off the list. She may pick a risk at random, or she may work through the list in some sort of orderly progression. Once she has a risk in mind, she starts to think about how the application she's testing could fail in ways similar to the risk. Having come up with ways in which her application could be subject to situations like the risk she's focusing on, she then thinks of tests that would show her whether those particular situations exist for the application.

[18] (Kaner, Falk, Nguyen, 1999)

For example, suppose Ellen is a tester responsible for testing a currency conversion program. This program updates its exchange rate data every 10 minutes from a central server. Suppose that Ellen then chooses the “Simple factual errors” risk off the list in *Testing Computer Software*. The description for this risk states “After a program changes, updating screen displays is a low priority task. The result is that much onscreen information becomes obsolete. Whenever the program changes visibly, check every message that might say anything about that aspect of the program.” After reading the description, Ellen brainstorms ways that this could apply to her application. She realizes that if a user had entered a conversion and left the result up, the next update of exchange rates from the server might not cause the result to update, even if there is a change in the rate used for the user’s calculation. She then starts thinking about tests that could show whether this happens or not, and decides to run a test where she does a conversion, then leaves the result up, goes to the server and changes the exchange rate for the currency she used, then wait for the update to occur back on her client machine, watching to see whether the calculation gets updated to match the new rate. In this case, Ellen used the risk idea as a heuristic to suggest a test case that she might not have otherwise tried.

Training Explorers

As mentioned at the beginning of this paper, the authors are working to determine how to most effectively train people to be exploratory testers. It is obvious to us that exploratory testing is not simply a matter of memorizing techniques. Exploratory testers need to draw heavily on their creativity, memory of past events, and intuition. While many people may be naturally gifted in these areas, that subset of the population is not the only group of people who can be exploratory testers. Exercises that encourage participants to develop their creativity, sharpen their memory, or focus their intuition will probably all play an important role. These exercises will draw on the extensive literature in other fields and tie these concepts into testing. One area in particular that we are considering is the similarities between training exploratory testers and training actors to excel in improvisational theater (or improv). When new students are brought into an improv class (and particularly when the students are not professional actors), they often need to learn how to express their creativity. A good improv actor also needs a memory of past events (since the majority of elements in a scene are imaginary, all actors in the scene have to remember the “reality” that has been created in the scene so far without visual cues to remind themselves). Intuition also plays a key role in improv. In his book, *Impro*, Keith Johnstone talks about having to train people to go with their intuition and their first thoughts when they perform an improvised scene, rather than second-guessing or censoring themselves. Obviously, there are parallels with exploratory testing, and the authors intend to investigate these parallels more fully.

Acknowledgements

We are grateful to the following people for the help they provided that led up to this paper:

- James Bach
- Sabrina Fay
- Elisabeth Hendrickson

- Heather Tinkham
- The participants of LAWST VII (Brian Lawrence, III, Cem Kaner, Noel Nyman, Elisabeth Hendrickson, Drew Pritzger, Dave Gelperin, Harry Robinson, Jeff Payne, Rodney Wilson, Doug Hoffman, James Bach, James Tierney, Melora Svoboda, Bob Johnson, Chris Agruss, Jim Bampos, Jack Falk, Hung Q. Nguyen, and Bret Pettichord)