

# Ad Hoc Software Testing

*A perspective on exploration and improvisation*

**Chris Agruss & Bob Johnson**

“Ad hoc, ad loc, and quid pro quo,  
So little time, and so much to know”  
- Jeremy in *Yellow Submarine*

## Abstract

*An ad hoc test can be described as an exploratory case that you expect to run only once, unless it happens to uncover a defect. As such, ad hoc testing is sometimes viewed as a wasteful approach to software testing. On the other hand, many skilled software testers find the exploratory approach to be one of the best techniques for uncovering certain types of defects. In this paper we'll put ad hoc tests into perspective with other forms of exploratory testing. Along the way, we'll also develop a comparison between jazz improvisation in the musical world, and improvisational software testing.*

## Introduction

You might wonder why we're placing such emphasis on the idea that ad hoc tests are fundamentally one-off cases. To begin with, the one-off characteristic distinguishes ad hoc tests cleanly from the formal testing paradigms that place the emphasis on re-executing tests, such as acceptance testing and regression testing. The assertion that one-off tests are valuable flies in the face of conventional wisdom, which places such a premium on repeatability and test automation. Much of the software industry overemphasizes the design and rerunning of regression tests, at the risk of failing to run enough new tests. It is sobering to ask why we are spending so much time rerunning tests that have already passed muster, when we could be running one of the many worthwhile tests that have never been tried even once?

In fairness, we understand that these issues are highly contextual. For example, if you're working with life-critical software, there are important ethical and legal reasons for placing such a heavy emphasis on regression testing. In general, though, we believe the software industry can benefit by striking a better balance between one-off tests, and those that are designed to be rerun. We'll return to this theme in the next section (*Ad hoc testing versus regression testing*), but first, let's take a look at what has already been written on this topic.

For various reasons, many people still associate ad hoc testing with an aimless black box approach. In practice, skilled testers use methods that are neither aimless, nor restricted to black box methods. The old notion of ad hoc testing is currently enjoying a metamorphosis into the more highly evolved approach called Exploratory Testing. We first came upon the term Exploratory Testing in the original edition of *Testing Computer Software* (Kaner, 1988). Here are a few excerpts:

“At some...point, you'll stop formally planning and documenting new tests until the next test cycle. You *can* keep testing. Run new tests as you think of them, without spending much time preparing or explaining the tests. Trust your instincts... In this example, you quickly reached the switch point from formal to informal testing because the program crashed so soon. Something may be fundamentally wrong. If so, the program will be redesigned. Creating new test series now is risky. They may become obsolete with the next version of the program. Rather than gambling away the planning time, try some exploratory tests – whatever comes to mind.”

More recently, Exploratory Testing has come into its own as a testing paradigm, as summarized on Brian Marick's superb Testing Craft web site: <http://www.testingcraft.com/exploratory.html>. In this excerpt, he lists the various elements associated with Exploratory Testing:

- An interweaving of test design and test execution. This is in contrast to a process in which the tests are all designed first, then run later.
- A notion that the tester is learning about the product by testing it.
- An emphasis on creativity and spontaneity.
- (Sometimes) An expectation that exploratory testing will change the allocation of effort. Testing thus precedes some parts of test planning.

James Bach has published a formal procedure for Exploratory Testing, available on the Testing Craft site mentioned above. One way he characterizes the Exploratory paradigm is that “the outcome of this test influences the design of the next test.” James also offers a seminar in Exploratory Testing, described on his website: <http://www.satisfice.com/seminars.htm>.

So how does the notion of an ad hoc test relate to Exploratory Testing? We assert that ad hoc testing is a special case of Exploratory Testing. In the course of doing Exploratory Testing, many of the test cases will be ad hoc (one-off tests), but some cases will not be. One way to distinguish between the two is to look at the notes associated with a given exploratory test. In general, exploratory tests have little or no formal documentation, but result in more informal notes. If the notes are detailed enough that the test could be rerun by reading them, then that is less likely to be an ad hoc test. Conversely, if there are no notes for a given exploratory test, or if the notes are directed more at guiding the testing effort than at reproducing the test, then this is almost surely an ad hoc test.

On the whole, software managers tend to view ad hoc testing skeptically. The notion of highly paid testers consuming precious time running one-off tests, and then discarding them, is apt to make managers squirm. However, skilled exploratory testers have little trouble justifying their time spent on ad hoc cases. The quantity and severity of defects unearthed using this approach can be astounding. In this paper, we’ll attempt to provide a useful context for this testing paradigm.

### ***Ad hoc tests versus Regression tests***

One aim of this paper is to propose a more precise definition for an *ad hoc test*. A standard definition for *ad hoc* is “for the particular end or case at hand without consideration of wider application” (Miriam-Webster, <http://www.m-w.com/>). Based on this, we will describe an *ad hoc test* as an exploratory case that you expect to run only once, unless it uncovers a defect. We’re aware that some people object to this definition of ad hoc testing, so if you’re in that camp, please feel free to substitute the term “one-off” for all instances of “ad hoc” in this paper. We prefer the term ad hoc, primarily for aesthetic reasons, but will use the two terms interchangeably in this paper.

We’ll try to put ad hoc testing into perspective here, by comparing it with other forms of testing. In particular, each method has strengths and weaknesses in the critical dimensions of *defect finding power* and *confidence building*. A primary goal of ad hoc testing is to uncover new defects in the product. In the hands of a skilled tester, it can be highly effective at discovering such problems. As a confidence builder, ad hoc testing is relatively weak, compared with formal regression testing, which can be a powerful confidence builder, especially if the breadth and depth of the coverage is demonstrably high.

The term *regression testing* has come to mean many different things, depending on the context in which it is used. For example, Cem Kaner has described these basic types of regression testing:

- Bug regression - try to prove that a newly fixed bug was not fixed.
- Old bugs regression - Try to prove that a recent code (or data) change broke old fixes.
- Side effect regression - Try to prove that a recent code change broke some other part of the program.

In the descriptions above, notice the emphasis on proving there are defects in the code, thus putting a premium on the defect finding power of regression testing. In contrast, consider this definition from the *whatis* online technical encyclopedia (<http://www.whatis.com/>):

“Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes. Regression testing is a normal part of the program development process and, in larger companies, is done by code testing specialists. Test department coders develop code test scenarios and exercises that will test new units of code after they have been written. These test cases form what becomes the *test bucket*. Before a new version of a software product is released, the old test cases are run against the new version to make sure that all the old capabilities still work. The reason they might not work is because changing or adding new code to a program can easily introduce errors into code that is not intended to be changed.”

In the definition above from *whatis*, notice the emphasis is placed on ensuring that the program is working, thus putting a premium on the confidence building aspect of regression testing. For the purpose of this paper, we’re going to use the definition from *whatis*, not because we think it’s a better approach than putting the emphasis on defect finding, but because we think this is closer to how the term *regression testing* is more commonly used today.

We’re also going to expand on this definition: implicit in the concept of creating a regression suite is the notion that the tests will be rerun periodically. Otherwise, there is little value in spending time creating the suite in the first place. Furthermore, many companies opt to automate their regression suite, specifically so that the tests can be rerun faster, less expensively, and more often. In this respect, regression testing is the opposite of ad hoc testing: an implicit yet important attribute of a regression test is that you expect to rerun it, in contrast to an ad hoc test, which you expect to run only once.

Although they are at the opposite ends of this spectrum, regression tests and ad hoc tests complement each other in remarkable ways. Ironically, when you uncover a defect with either form of testing, you’ll be drawn toward the opposite end of the spectrum. When you find a defect with a regression test, you’ll often need to use ad hoc methods to analyze and isolate the defect, and to narrow down the steps to reproduce it. You may also want to explore “around” the defect to determine if there are related problems. On the other hand, when you find a defect with an ad hoc test, you’ll probably document it in your defect tracking system, thereby turning it into a regression test, for verification after the defect is fixed.

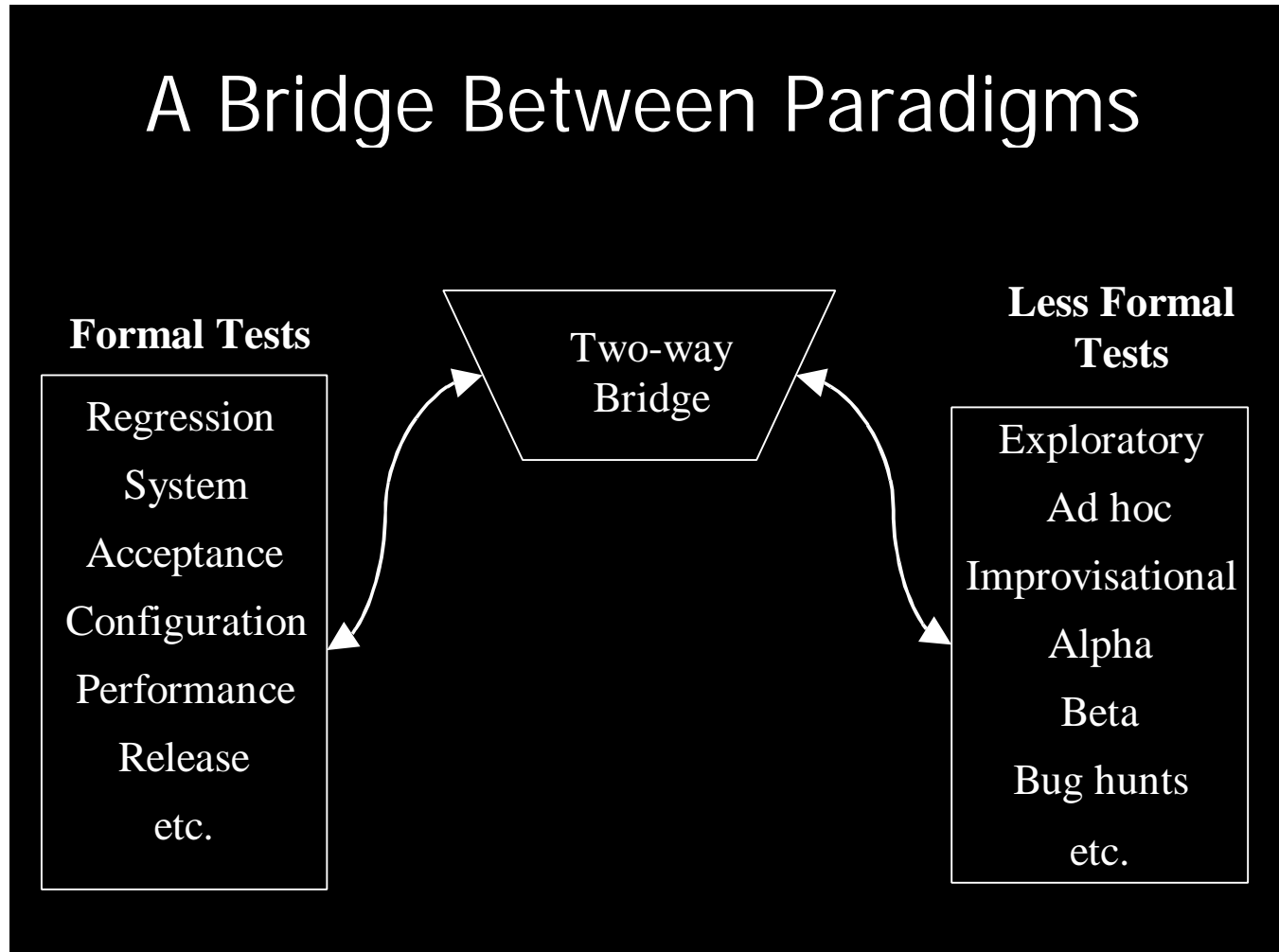
Let’s consider some other specific types of testing along this spectrum. For the sake of visualization, we’ll place fully repeatable regression tests on the left end of the spectrum, and ad hoc tests on the right end. For any given method used by a tester, its location along this spectrum will vary, depending on the methods being used by the tester. For example, performance testing and release testing have a dominant regression testing component, so we would place them along the left side of the spectrum. Exploratory testing and User Scenario testing have a dominant ad hoc component, so we would place them somewhere on the right side of the spectrum.

We’re convinced that most software testing projects will benefit from an intelligent combination of the various forms of testing. Rare is the project that can be tested optimally by regression testing or by ad hoc testing alone, but by finding the ideal proportions for each project, you’ll have a better balance between *defect finding* and *confidence building*.

### ***The Metaphorical Bridge***

The spectrum described above can also be pictured as a bridge between formal and less formal testing paradigms. This metaphor has the advantage of suggesting motion across the bridge in both directions. For example, you can bring regression tests across the bridge into the ad hoc domain for more extensive testing. Conversely, you can

bring ad hoc tests that you choose to rerun across the bridge in the other direction, for integration with the regression suite. We'll expand this bridge metaphor more fully in the section on Improvisational Testing, below.



## ***Improvisational Testing***

One approach to ad hoc testing is to treat it as improvisation on a theme. This is similar in some ways to jazz improvisation in the musical world. Testers often start with a documented Test Design that systematically describes all the cases to be covered. Similarly, jazz musicians sometime use a *fake book* consisting of *lead sheets* for the songs on which they will improvise (for an example of a fake book, see: <http://www.pianospot.com/1701053.htm>). After playing the recognizable melody (the *head*) once or twice, the musicians will take turns playing extemporaneous solos. Sometimes they will also vary the rhythm of the piece while improvising; for example, by playing behind the beat. These improvisational solos may be recognizable as related to the original tune, or they may not. However, toward the end of the song, the players typically return to the original melody.

Another aspect of jazz, of course, is that musicians usually perform together, with musicians often taking turns playing the lead. In the world of software testing, there's an intriguing parallel. One of the more productive ways to perform improvisational testing is to gather a group of two or more skilled testers in the same room, and ask them to collaborate on extemporaneous testing. If you provide the collaborators with individual machines with which to test in this circumstance, there's often a mixture of ensemble and solo efforts that emerge. On the basis of anecdotal evidence, the defect finding power of people collaborating in this way is considerably greater than the sum of its parts, so there appears to be a high potential for group synergy with improvisational testing. Indeed, while editing this very page, two testers in the next cube found a defect and added two new use cases to our documentation.

As mentioned earlier, one way to approach improvisational testing is by using existing documented tests as the initial theme, and then inventing variations on that theme. For example, imagine that you're testing an image-editing program named Panorama (a fictitious name) that stitches together a series of photographs into a seamless panoramic image (see <http://www.thams.com/desert/dhp.72.med.jpg> for examples of a few such photos).

Perhaps you have a documented series of tests that would lead you in some fashion through the following steps:

1. Load the Panorama program
2. Import 8 photographic images that together represent a full 360 degree panorama
3. Invoke the auto-stitching feature
4. Verify that the auto-stitching sequence completes without errors
5. Verify that the resulting image is seamlessly stitched into a 360-degree vista
6. If any of the stitches are not aligned correctly, invoke the manual stitching feature
7. Manually align any stitches that look incorrect
8. Invoke the final stitch feature
9. Verify that the final stitch feature completes without errors
10. Verify that the resulting panorama looks correct.

Looking at this series of tests, one thing you might notice is that the verification steps would be difficult to automate. The desired result is an aesthetically pleasing and perceptually convincing panoramic image, but there may be many different resulting images that would pass the test. Assuming that you'll be running these manually, what are some useful variations you could introduce for these tests?

For starters, notice that the 10 steps listed above contain a repeating pattern: 3 execution steps, then 2 verification steps, followed again by 3 execution steps and 2 verification steps. Testers often fall into patterns like this without even realizing it, which limits the types of tests they design. Therefore, one simple way to introduce a variation would be to alter that pattern. This could be done in any number of ways, such as inserting new steps into the testcase that should not change the end result. For example, you could:

- run step 3 (auto-stitching) two or more times in succession before proceeding to step 4
- manually align the stitches first, prior to running the auto-stitch algorithm, and then proceed as before
- between any two steps, toggle your system's display color depth settings (the number of colors displayed) back and forth before proceeding.

As you can imagine, the possibilities are nearly endless. However, what makes Improvisational Testing a craft is the desire to narrow the field to the cases that are most likely to uncover problems. This is where experience makes all the difference. You can find more ideas on this later in the Techniques section of this paper.

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 explore “around” the fix, inventing variations on the steps to reproduce the original defect, ensuring more fully that the fix didn't have undesirable side effects elsewhere.

If you're unfamiliar with this concept of testing around a defect, here's an example, using the Panorama test scenario above. Let's say that you've found a defect such that when you manually adjust any of the stitches, all the stitches to the left of the one you've adjusted become misaligned. After the defect is fixed, you verify that all the seams to the left are now behaving properly. In addition to this, you could also improvise some additional tests, such as verifying that the stitches to the right of a manually aligned stitch still behave properly too.

Returning to the bridge metaphor, improvising in this way is an example of crossing the bridge from the formal toward the less formal testing methods. Let's imagine that from the currently existing documented test you invent

ten new variations on that case through improvising. Of those ten, perhaps one of them uncovers a new defect, eight of them are run as one-off tests (we'll discard those after running them once), and one case you decide to add to your regression suite. In the case of the two new tests that you will be rerunning (the defect, and the one you added to the regression suite), you have made the decision to carry them back across the bridge in the other direction.

This scenario raises the question, "How do you decide whether a one-off test should be archived, to become part of your documented regression-testing suite?" The answer to this depends heavily on your situation, but here's are some factors to consider. Assume for a moment that you have run this test one time already, and that it has passed the test. Next, make an educated guess as to what the probability is that rerunning this test in the future will uncover an important defect. Some factors you could use to make this decision include:

- Does this feature reside in a relatively unstable area of the program?
- Can this test be automated easily, so that it can be rerun many times at lower execution cost?
- If this test were to fail, what would be the severity level of the defect?

From the discussion above, you may have noticed one way in which improvisational tests differ from ordinary ad hoc tests. With ad hoc tests there is no expectation to rerun them unless a defect is found, but with improvisational tests, we're actively looking for cases that would be appropriate to preserve and rerun as part of our regression suite. For this reason, we would place Improvisational Testing closer to the middle of the spectrum between regression testing and ad hoc testing.

### ***When is Ad Hoc Testing NOT Appropriate?***

We've seen attempts at using ad hoc methods as an adjunct to build acceptance testing, with mixed results. Our opinion is that ad hoc methods are more appropriate elsewhere. If a testing task such as build acceptance needs to be done repeatedly, it should be documented, and then automated. We like to include the build regression tests in the make file, so that there is no confusion about running these tests. This has also encouraged developers to add more tests.

Neither do release tests lend themselves well to ad hoc methods. Release protocols need to be documented and executed meticulously, to assure that every critical item is checked off the list. These tests need to be complete and highly organized. If release testing fails, the product will need to be fixed, and the tests rerun. Often, this is a driving force behind having release tests automated. Moreover, these tests usually need to be run quickly, whereas ad hoc testing is typically a slower, more labor-intensive form of testing.

Crash parties and bug hunts may be good team building or marketing events, but these events rarely result in the sort of ad hoc testing that we're advocating. It might be exciting to have your CEO sit down and try to break the program, but it is also very risky, unless the program is thoroughly tested beforehand. We have seen such events go badly, particularly if the CEO breaks the program and everybody goes back to work feeling demoralized.

### ***Strengths of Ad Hoc Testing***

One of the best uses of ad hoc testing is for discovery. Reading the requirements or specifications (if they exist) rarely gives you a good sense of how a program actually behaves. Even the user documentation may not capture the "look and feel" of a program. Ad hoc testing can find holes in your test strategy, and can expose relationships between subsystems that would otherwise not be apparent. In this way, it serves as a tool for checking the completeness of your testing. Missing cases can be found and added to your testing arsenal.

Finding new tests in this way can also be a sign that you should perform root cause analysis. Ask yourself or your test team, "What other tests of this class should we be running?" Defects found while doing ad hoc testing are often examples of entire classes of forgotten test cases.

Another use for ad hoc testing is to determine the priorities for your other testing activities. In our example program, Panorama may allow the user to sort photographs that are being displayed. If ad hoc testing shows this to work well, the formal testing of this feature might be deferred until the problematic areas are completed. On the other hand, if ad hoc testing of this sorting photograph feature uncovers problems, then the formal testing might receive a higher priority.

We've found that ad hoc testing can also be used effectively to increase your code coverage. Adding new tests to formal test designs often requires a lot of effort in producing the designs, implementing the tests, and finally determining the improved coverage. A more streamlined approach involves using iterative ad hoc tests to determine quickly if you are adding to your coverage. If it adds the coverage you're seeking, then you'll probably want to add these new cases to your archives; if not, then you can discard the cases as one-off tests.

There are two general classes of functions in the code that forms most programs: functions that support a specific feature, and basic low-level housekeeping functions. It is for these housekeeping functions that ad hoc testing is particularly valuable, because many of these functions won't make it into the specifications or user documentation. The testing team may well be unaware of the code's existence. We'll suggest some methods for exploiting this aspect of ad hoc testing in the Techniques section below.

### ***Techniques for Ad Hoc Testing***

So, how do you do it? We are reminded of the Zen aphorism: those that say do not know, and those that know do not say. This is a challenging process to describe, because its main virtues are a lack of rigid structure, and the freedom to try alternative pathways. Much of what experienced software testers do is highly intuitive, rather than strictly logical. However, here are a few techniques that we hope will help make your ad hoc testing more effective.

To begin with, 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, with relatively little attention paid to the potential interaction between features. Much of this interaction goes on at the subsystem level (e.g. the graphical subsystem), because it supports multiple features. Picture 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. In our Panorama example, this might be the interaction between the "stitch" feature and the "import" photograph feature. Can you stitch and then add more pictures? What are all the possibilities for interaction between just these two features?

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, 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 such problems.

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 Panorama's garbage collection (random access memory cleanup and reuse). From the documentation given, you will probably be uncertain if any garbage collection testing is required. 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. Some other areas to consider include:

- sorting
- searching
- two-phase commit
- hashing
- saving to disk (including temporary and cached files)
- menu navigation
- memory management (beyond garbage collection)
- parsing

A good ad hoc tester needs to understand the design goals and requirements for these low-level functions. What choices did the development team make, and what were the weaknesses of those choices? As testers, we are less concerned with the correct choices made during development. Ad hoc testing can be done as black box testing. However, this means you must check for all the major design patterns that might have been used. Working with the development team to understand their choices moves you closer to clear or white box testing. This allows you to narrow the testing to many fewer cases.

Read defect reports from many projects, not just from your project. Your defect database doesn't necessarily 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, and 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 while running one-off tests. In many cases you won't see a visible error in the execution of the program, yet one of the tools will flag a process that is out of control.

### ***Where does ad hoc testing fit in the testing cycle?***

Ad hoc testing finds a place during the entire testing cycle. Early in the project, ad hoc testing provides breadth to testers' understanding of your program, thus aiding in discovery. In the middle of a project, the data obtained helps set priorities and schedules. As a project nears the ship date, ad hoc testing can be used to examine defect fixes more rigorously, as described earlier.

### ***Can Ad Hoc Testing Be Automated?***

Regression testing is often better suited than ad hoc testing to automated methods. However, automated testing methods are available that provide random variation in how tests are executed. For example, see Noel Nyman's excellent paper listed in the reference section: GUI Application Testing With Dumb Monkeys. Yet, a key strength of ad hoc testing is the ability of the tester to do unexpected operations, and then to make a value judgement about the correctness of the results. This last aspect of ad hoc testing, verification of the results, is exceedingly difficult to automate.

Partial automation is another way in which automated testing tools and methods can be used to augment ad hoc testing in particular, and exploratory testing in general. Partial automation implies that some of the test is automated, and some of it will rely upon a human. For example, your ad hoc testing may require a great deal of set up and configuration, which could be covered by an automated routine. Another commonly used scenario involves designing an automated routine to run a variety of tests, but using a human being to verify whether the tests passed or failed. For logistical reasons, this last method is more feasible for low volume tests than it is for high volume approaches.

There are yet other ways in which automation tools can help the manual ad hoc tester. Ordinarily, we advise against using record/playback tools for automating test cases, but a good recording tool may help in capturing the steps in a complex series of ad hoc operations. This is especially important if you uncover a defect, because it will bring you that much closer to identifying the steps to reproduce it. Some testers have told us that they routinely turn on their recorders when they begin ad hoc testing, specifically for this purpose. Alternatively, you could opt to place tracing code in the product. When isolating a defect, tracing code may provide more useful info than a captured script will. Tracing can also help you understand how the lower level code, your target for subsystem testing, is being used.

### ***Future Directions***

Some of the most interesting new work in this area is coming from Cem Kaner, a Professor at the Florida Institute of Technology. From looking over his course notes, its clear that Cem has collected a rich set of techniques that can help guide the exploratory tester.

Brian Marick has described a somewhat different form of improvisational testing in his article *Interactions and Improvisational Testing* - <http://www.testingcraft.com/exploiting-interactions.html>. In that article, he describes Improvisational Testing as “testing that's planned to actively encourage imaginative leaps.” This appears to be another very fruitful dimension to improvisational testing that warrants further exploration.

### ***Acknowledgements***

Thanks to Brian Lawrence, for reviewing this paper early on, and to Brian Marick, who reviewed it in its latest form. Additionally, we wish to thank all the folks from the Los Altos Workshop on Software Testing (LAWST) for the stimulating conversations from which some of these ideas emerged.

On July 10<sup>th</sup> and 11<sup>th</sup> of 1999, twenty experienced software testers met at the Seventh Los Altos Workshop on Software Testing (LAWST VII) to discuss Exploratory Testing. This paper was originally presented the year before that, but revisions have been made in the current version to reflect some of the insights from that meeting. A hearty thanks to all attendees.

LAWST VII Attendees:

- Brian Lawrence & III (facilitators)
- Hung Nguyen
- Jack Falk
- Bret Pettichord
- Melora Svoboda
- James Tierney
- James Bach
- Doug Hoffman
- Rodney Wilson
- Jeff Payne
- Harry Robinson
- Dave Gelperin
- Drew Pritzger
- Elisabeth Hendrickson
- Cem Kaner
- Noel Nyman
- Chris Agruss
- Bob Johnson

## ***Suggested Reading***

We recommend starting with Knuth's [The Art of Computer Programming](#), especially volumes 1 and 3. Another source of knowledge for the ad hoc tester is Webster's [Pitfalls of Object-Oriented Development](#). For more on how to approach testing software subsystems, see Brian Marick's [The Craft of Software Testing](#). Find out all you can about the full range of mistakes that designers and programmers make, and then go see if your project has fallen into any of these pits.

## ***References***

Bach, James, *General Functionality and Stability Test Procedure*, <http://www.testingcraft.com/bach-exploratory-procedure.pdf>.

Kaner, Cem, Jack Falk, and Hung Nguyen, [Testing Computer Software](#), 2<sup>nd</sup> Edition, John Wiley & Sons, 1999.

Knuth, Donald, [The Art of Computer Programming](#), Volume 1, Fundamental Algorithms, Addison Wesley Publishing Company, 1973.

Knuth, Donald, [The Art of Computer Programming](#), Volume 3, Sorting and Searching, Addison Wesley Publishing Company, 1973.

Marick, Brian, [The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing](#), Prentice Hall, 1994.

Nyman, Noel, *GUI Application Testing With Dumb Monkeys*, Proceedings of STAR West, 1998.

Webster, Bruce, [Pitfalls of Object-Oriented Development](#), M&T Books, 1995.

## ***Biographies***

Bob Johnson has spent 23 years developing and testing software. As a Senior QA Engineer, he's been involved in both defining the test strategy and training new staff in testing commercial off-the-shelf software.

Chris Agruss has been testing software since 1984, with a focus on test automation since 1994. He manages several small teams of testers and developers for Discreet (<http://www.discreet.com>).

Copyright © 2000, Bob Johnson and Chris Agruss. All rights reserved.