Extracted from:

Explore It!

Reduce Risk and Increase Confidence with Exploratory Testing

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

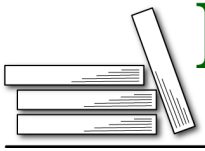
The Pragmatic Programmers

Explore It!

Reduce Risk and Increase Confidence with Exploratory Testing

Elisabeth Hendrickson

Edited by Jacquelyn Carter



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-02-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2013

It is easy to see how the techniques for exploring a system apply when there is a user interface to manipulate. You can see and manipulate fields and controls. Sometimes it is a little more difficult to see how to apply these techniques on systems that don't have a GUI: servers, APIs, embedded systems, and batch programs.

The techniques in this book apply to absolutely any kind of software.

None of the analysis techniques require a user interface to make things visible. Consider the most basic analysis technique: identifying things to vary. You can always find interesting things to vary even if they are not exposed in a user interface. Imagine a batch processing program: it reads data from a file, transforms it in some way, and sends it on to another part of the system. You could vary the data in the file, the size of the file, or the count of records in the file.

Further, heuristics like Zero, One, Many are abstract building blocks that work regardless of the interface you use to manipulate the system. Consider that hypothetical batch processing system again. You can load in a file with zero, one, or many records. You can violate the domain-specific rules for the values in fields in the file. You can leave fields blank. Indeed, I have seen bugs triggered by each of those conditions in real-world batch processing systems: programs that crashed if the file was empty, if any of the fields were empty, or if any fields contained unexpected data.

Any system has at least one kind of interface, and usually more. The interface isn't always for humans. Sometimes it's to connect with other systems. It's still an interface, and that means you can still use it to explore.

In this chapter you'll see four examples of exploring non-GUI interfaces: a Java API, a JavaScript function, a web service using XML, and a server.

10.1 Exploring an API

I had just completed a session on exploratory testing at an internal conference for a company. One of the programmers in the audience approached me after the session. Let's call him "Colin."

"So," he began as he approached me, "this exploratory stuff sounds good, but it doesn't apply to what I do. I write low-level code, APIs that other programmers use. It looks to me like exploratory testing is only relevant to GUI-level stuff."

I immediately regretted doing my demo with a web application, and I knew that no amount of explaining now would erase the image of a JavaScript-centric, graphically rich interface from his mind.

"I don't think that's true," I replied. "How about if we pair on something you're working on right now and see if the heuristics I just went over could apply to your code?"

Colin agreed and led me over to a computer.

"Here's a little side project I'm working on," he said as he brought up his editor. "It's a library with functions for processing text." He pointed to a particular section of code. "This is a function that can compare two strings and score their similarity. It implements a scoring algorithm derived from cosine similarity and returns a value between 0 and 1,[1] where 0 means the two strings are not at all similar and 1 means the strings are a 100 percent match."

My mind automatically started analyzing the variables. I looked at the signature of the function we would be exploring. It took two, and only two, string parameters:

```
public double calculateSimilarity(String stringA, String stringB)
```

Because Java is a strongly typed language, the Java compiler would enforce the type and number of parameters. That meant that I could not vary the type of data I used in calling the function. For example, calling it with integers would be an invalid test:

```
calculateSimilarity(3, 5); // invalid because integers aren't strings
```

And I could not explore it by varying the number of parameters I passed in, so the Zero, One, Many heuristic would not apply to the number of parameters:

```
calculateSimilarity();    // invalid because it requires two parameters
```

More accurately, I could vary these things, but then I would be exploring the Java compiler and not the function we had set out to explore. So I mentally discarded these tactics and considered what I could vary with two strings.

The length of each of the strings was a variable, as were the characters in the strings. The similarity and difference between the lengths and characters represented even more variables. Finally, the output score was a variable. I decided to focus on the output first and suggested that we find ways to get scores of 0 and 1.

1. http://en.wikipedia.org/wiki/Cosine_similarity

Together, Colin and I wrote a little program that called the function and returned the result. We tried values that we were sure would be a perfect match, resulting in a score of 1:

```
calculateSimilarity("a", "a");
```

The function returned a 1 as expected. Next we changed our program so that it called the function with two completely different values.

```
calculateSimilarity("a", "z");
```

The function returned 0, again as expected. Colin was starting to look a little bored; these tests were too simplistic. He didn't realize I was just starting. I modified the program to call the function with two zero-length strings.

```
calculateSimilarity("", "");
```

The function returned 1, a complete match. So I tried with null values:

```
calculateSimilarity(null, null);
```

The program threw a Java exception and exited. Colin made a note.

Next I experimented with the length of the input strings. I typed a paragraph worth of nonsense and assigned it to a variable:

```
String myString = "a very long paragraph ...
calculateSimilarity(myString, myString);
```

(My actual paragraph was about a hundred words long.)

The program ultimately returned the expected value of 1, indicating that the two strings were an exact match (as they should be since they were exactly the same string). However, I noticed that the program paused significantly before returning the results.

So I decided to try comparing two very long strings. I opened a browser to Gutenberg.org and copied the first chapter from Mark Twain's classic book *Tom Sawyer*.[2] I pasted the value into a variable and compared it against itself again. We waited. Nothing seemed to be happening. I opened a system monitor. The computer's CPU was pegged at 100 percent. Something was happening; we just couldn't see what.

Colin took another note.

"Looks like there's a performance issue," I observed.

2. http://www.gutenberg.org/ is an excellent source of long passages of text.

Colin nodded. “And it would be a huge problem if I wanted to use this to detect plagiarism,” he said. “Then I’d be comparing a student’s paper against multiple sources, and it would just take too long to process.” I glanced over. He looked thoughtful.

“We could kill the process and try some more experiments,” I suggested. “I’d like to see how it handles other kinds of text, like accented characters, Japanese or Chinese characters, nonprinting characters, and different kinds of whitespace.”

Colin laughed. “I don’t think you need to. You convinced me,” he said. “Exploratory testing is most definitely not just about GUIs. I’ll explore more later. Let’s go back to the conference now.”

10.2 Exploring a Programming Language

As a programmer, it’s important to understand the ins and outs of whatever language and libraries you use. Misunderstanding how the underlying technology behaves is a sure recipe for creating bugs.

In his comically genius “WAT” video,[3] Gary Bernhardt demonstrates how illogical JavaScript and Ruby can be. It’s a fantastic example of exploring programming languages to discover quirks and surprises that can bite the unwary programmer.

Inspired by the video, I decided to explore the JavaScript sort() function. I began with a simple case:

```
["b", "c", "a"].sort()
```

This returned the results I expected:

```
a,b,c
```

So far so good. I began brainstorming things I could vary: the count of items in the array (Zero, One, Many), characters in strings (contents and character sets), and the types of objects (strings, numbers, objects, arrays).

First I tried varying the count of items to be sorted. Here are the conditions I tried and the corresponding results:

```
> [].sort()


> ["a"].sort()
a

> Array(99999).sort()
```

3. https://www.destroyallsoftware.com/talks/wat/

```
,,,,,,,,,,, // ... and lots more commas
```

There was nothing particularly interesting there. So I turned my attention to data types. I decided to try numbers:

```
> [7, 3, 11].sort()
11,3,7
```

Whoa! JavaScript wasn't sorting numbers as numbers. It sorted the numbers as characters. 11 is numerically greater than 7 but alphabetically before 3. A quick search on the Internet revealed that if you want something other than an alphabetical sort, you have to pass in a function, like so:

```
> [7, 3, 11].sort(function(a,b){return a-b})
3,7,11
```

This is exactly the kind of surprise I was looking for, a language quirk that can lead to surprisingly bad behavior if you don't know about it. Worse, because alphabetic and numerical sorts only yield different results if the numbers have a different number of digits, it's the kind of subtle thing that's hard to pin down.

Moving on, I decided to experiment more with data types. Arrays in JavaScript can hold any type and can even mix types. So it's perfectly valid to have an array that contains both numbers and letters. So I tried this:

```
> [1.1, 0, "a"].sort()
0,1.1,a
```

I was back to things that were not all that interesting. What about arrays, objects, and special values?

```
> ["a", {"foo": "bar"}, Infinity].sort()
Infinity,[object Object],a
```

Now things were getting interesting again. Infinity was smaller than an object? A little more experimenting showed that once again the results were coming out in alphabetic order. *I* comes before *O* in the alphabet, so Infinity is smaller than Object. What if I passed in the numeric compare function again?

```
> ["a", {"foo": "bar"}, Infinity].sort(function(a,b){return a-b})
a,[object Object],Infinity
```

The key lesson learned here is that no matter what kind of things you have in an array in JavaScript, the default behavior is to cast them to a string and then sort alphabetically. If you want different behavior, you have to override the sort function.

The other thing that is important to take away from this example is that exploration isn't just for production code. Exploratory techniques can help you to characterize the capabilities and limitations of the technology on which the code is built.