CHAPTER 4

Find Interesting Variations

Once upon a time, I was interviewing a tester. I showed him a screen mockup and asked him what kinds of things he would want to test. He listed four or five scenarios he'd try. Then he leaned back in his chair and nodded to himself. "Yup," he said, "That'd do it."

He didn't get the job.

It's not that the four or five things he listed were somehow lacking. They weren't. He had good ideas. The problem was that he was completely satisfied with his answer. He didn't ask questions. He didn't even leave open the possibility that there was something he hadn't considered. He signaled clearly that he was done.

Even in a relatively simple system, there are an infinite number of variations to explore, including user interactions, sequences, timing, data, configurations, and environmental factors like network traffic or CPU load. That's why there's an old joke that "exhaustive testing" is when the tester is too exhausted to continue.

Given the infinite landscape to explore, being completely satisfied with a bare handful of experiments is downright dangerous. If you explore only superficially and then report that all's well, as I feared this interview candidate might, your stakeholders will make bad decisions, not realizing they are missing a big piece of the picture.

Good explorers recognize that there are an infinite number of variations and they cannot possibly cover all of them. Great explorers know how to hone in on the most interesting variations. There is a skill in analyzing a system to discover novel and yet realistic ways to manipulate it.

This chapter is about recognizing things to vary. Actually, because any testing effort involves varying how you exercise the software, much of this book is

about discovering interesting variations. However, this chapter in particular focuses on helping you learn to see those variations. You'll learn to see past the superficial, obvious things to find subtle or hidden variations that are overlooked all too often. You'll find ways to structure your search for interesting variations and to connect those variations with the techniques in later chapters. Above all, you will expand your thinking, prompting you to consider new dimensions for exploring your software.

4.1 Variables Are Things That Vary

This chapter is about variations, and a variable is the indivisible atomic unit of variation. Because the word *variable* comes with some baggage, this might require some explanation. If you have ever written a line of code, you might think of a variable as a named location in memory that you declare with statements like this:

```
int foo;
```

That's great knowledge; it's what a variable looks like inside a program. However, when you're exploring software, you're manipulating a different kind of variable.

> In testing, a variable is anything that you can change or cause to be changed indirectly while operating the software.

In particular, you want to find the variables that might affect the system behavior in interesting ways. There are three different kinds of variables to consider: the obvious ones that jump out at you, the subtle ones that are easy to miss, and the ones that are only indirectly accessible.

Obvious Variables

Some variables are immediately obvious. For example, if you are exploring a GUI that has fields on a form, the fields are variables because you can change their values. If you're exploring an API, the values you pass into the API calls are variables. Although such obvious variables are important, they're often the least interesting because they've typically already been tested extensively.

Although simply changing the values might not be that interesting, you can certainly choose interesting values. Different values exhibit different characteristics.

For example, consider software that takes in a binary input (a sequence of 0s and 1s) and converts it to a decimal value. If you input the value "101" or "011," you might expect it wouldn't make much difference. However, "101" is symmetric; "011" is not. If you choose an input value that is symmetric,

you won't notice if the program is reading the bits in the wrong order. By contrast, you'll most likely notice if the program interprets the binary number "011" as equal to the decimal value 6 instead of the decimal value 3.

Thus you see that variables are fractal in nature. Having found one thing to vary, such as the value of an input, you discover that you can also vary aspects of the values you choose, such as the symmetry.

Subtle Variables

The symmetry of an input is an example of a very subtle variable. Other subtle variables may be visible, but they are not intended to be changed directly, like parameters in a URL in the address bar for a browser. For example, imagine you're exploring a website and notice that the browser shows the address of the current page as this:

```
http://example.com?page=3&amp;user=fred
```

There are two variables after the ? character in the URL: the page number and the username. They're contained within the key/value pairs in the URL string.

Theoretically you're not supposed to manipulate such variables directly; they're set by the web application when you click a link. However, people do mess with URLs. Sometimes users change the URL for a good reason: they want to hop directly to a different page in a report or change a query without having to navigate through the UI. Or perhaps they have a stale bookmark. Other times malicious users are trying to find a security vulnerability.

So what happens if a page or user doesn't exist in the system? In some systems users see an ugly error.

Or what if they hack the URL so that it contains something like "user=foo;'drop table customers;"? In an insecure system, users manipulating the parameters passed to the server can cause very real damage. So exploring for these kinds of vulnerabilities is a good idea, especially when you find variables that weren't intended to be changed by users.

Another example of settings that users aren't supposed to change directly can include hidden preferences settings. Sometimes there are settings in cookies or configuration files that are supposed to be set only by the software. If users change these values, they may gain access to something they shouldn't. So as you're exploring to discover variables, look for hidden settings as well.

Indirectly Accessible Variables

Some of the most interesting variables are buried deep. They're things that can only be controlled indirectly, like the number of users logged in at any given time, the number of results returned by a search, or the occurrence or absence of a condition.

These variables are easy to miss, but if you can find them and exploit them, they often reveal critically important information about the system, as you'll see in the next section.

4.2 Subtle Variables, Big Disasters

Let's consider some famous software failures to see how important, and how difficult, it can be to identify subtle variables.

The Therac-25 Case

In the late 1980s, a number of cancer patients treated with the Therac-25 radiation therapy machine sickened after their treatment. Some died. The cause of death was radiation poisoning.

Investigation revealed that under certain circumstances the machine delivered an overdose of radiation rather than the safe medical dosage that the doctor prescribed. The investigation team concluded that the root cause was not a single failure but rather a complex set of interacting variables. Although hardware safety deficiencies contributed to the malfunction, the software had serious defects worth examining.

As Nancy Leveson explains in her book *Safeware [Lev95]*, at least one of the incidents happened because the technician entered and then edited the treatment data very quickly, in less than eight seconds. That eight seconds also happened to be the time it took the magnetic locks to engage, creating a short but devastating window of vulnerability.

The speed of input is a very subtle variable. The way users interact with the system is always a variable, of course, but the speed of input, and the fact that there was an important threshold at the eight-second mark, is quite subtle.

Yet it's also a potentially common situation: power users can generally manipulate systems with astonishing fluency. Using keyboard shortcuts and with lightning fast typing speeds, they may be able to enter data so quickly that they get ahead of the system. In response, the system may exhibit interesting behavior, such as unexpected errors or, as in this case, catastrophic failure.

Further, Leveson found that every 256th time the setup routine ran, it bypassed an important safety check. The number of times you've started a system is always interesting. Frequently systems do things differently the first time you run them. In this case the system worked differently the 256th time, the 512th time, and so forth, that it was booted up. That difference in behavior meant the system was vulnerable to potentially fatal malfunctions.

The Ariane 5

In 1996, the Ariane 5 rocket exploded spectacularly during its first flight. Investigation revealed that the root of the problem was a conversion of a 64-bit floating-point number with a maximum value of 8,589,934,592 to a 16-bit signed integer value with a maximum value of 32768. That conversion caused an overflow error. Compounding the problem, the system interpreted the resulting error codes as data and attempted to act on the information. As a result, the rocket veered off course. When it detected the navigation failure, it self-destructed as designed.

The conversion problem stemmed from differences between the Ariane 5 rocket and its predecessor, the Ariane 4. The control software was originally developed for the Ariane 4. However, the Ariane 5 rocket was significantly faster than the Ariane 4, and the software simply could not handle the horizontal velocity the rocket sensors registered.

Velocity is the key variable in this case. For a rocket, it's kind of an obvious variable. The tricky bit, and the less obvious detail, was that there was such a difference in the horizontal velocity for the Ariane 4 and the Ariane 5. So the platform on which the software is running is another key variable. If you have ever done cross-browser or operating system compatibility testing, you know how critical the platform on which the software is running can be.

The Mars Rover

In 2004, NASA lost contact with the Mars rover Spirit. NASA soon discovered that Spirit had encountered a serious anomaly and was rebooting itself over and over again. For a while, it looked like the rover's mission was over before it really began.

An article in *Spaceflight Now* explains that the problem was caused by the number of files in flash memory.[1] The rover had started collecting data on its flight to Mars. Some operations created numerous small files. Over time, the table of files became huge. The system mirrored the flash memory contents

1. http://www.spaceflightnow.com/mars/mera/040201spirit.html

in RAM, and there was half as much RAM as flash memory. Eventually the table of files swamped the RAM, causing the continuous reboots.

The key variable here isn't just the amount of disk space used; it's the number of files. Testing this by filling up the available space with a single giant file might not reveal the same behavior as having lots and lots of little files. So even when we talk about variables like space available on the disk, subtle details matter.

In exploring any system, there are variables and then there are variables within variables. It's variables all the way down. Becoming adept at taking note of interesting variables is the key skill that will enable you to explore absolutely any kind of software.

In the next section, you'll learn what to look for to spot subtle variables and how to manipulate them.

4.3 Identifying Variables

Until you become accustomed to looking for variables, it can be difficult to spot the subtle and inaccessible variables. The trick is to use common patterns of variables to help you learn to see the things you can vary in your software. This section offers a list of kinds of variables to watch for in your software.

Things You Can Count

Every system has things you can count. It might be a count of user accounts in the system or the number of times an account is logged in. It could be the number of phone numbers associated with a profile or the number of printers configured on a machine. Perhaps it's the number of files to be processed in a batch run or the number of records in a given file.

Counts are a subtle variable often overlooked until it's too late. Notice that two of the three stories above deal with counts: the Mars rover story hinged on having too many files, and one of the risk factors associated with the Therac-25 malfunctions was the number of times the setup routine had been run.

Once you find interesting things to count in your system, you can use these heuristics to vary the counts.

- *Zero*, *One*, *Many*: Do a search to return zero, one, and many records, and you may notice a common gaffe: messages that say "0 record found" or "1 records found."

- *Too Many*: Create conditions with more things than the software can handle, such as too many open connections or too many requests.
- *Too Few*: Create conditions with fewer things than the software expected, such as too few items to populate a list or too few line items on an invoice.

A count of zero is often interesting in its own right. Often software that is expecting a set of things does not handle the case where the set has no elements. Thus, zero is a heuristic in its own right.

For example, a desktop application I tested behaved badly if the computer had zero printers installed. For a more recent example, consider Chess Wars, an iPhone game designed to allow you to challenge your Facebook friends to a game of chess. When you tried to find someone to challenge, if zero of your Facebook friends had the Chess Wars app, your iPhone app exited unexpectedly.[2] When the application was first released, only the few people involved in creating it or beta testing it had it installed, so nearly all new users encountered this bug.

Relative Position

When you identify situations where things have a relative position, you can apply the Beginning, Middle, End heuristic.

For example I once tested a text editor. Pasting text at the very end of a line caused problems, but pasting text at the beginning or middle did not.

On another system that presented data in a list, deleting the very last item in the list always failed. You could delete the first item or items from anywhere in the middle of the list, but you could not delete that last entry.

You can also explore areas where you can vary position to discover if items take the correct position relative to the other items. You might have a stack of objects on top of one another, like elements in a drawing with a z-order determining which appear at the front or the back. You could vary the position of the elements from front to back to see if they are still rendered correctly. If you have a sortable list, you can create items that should sort to the top or the bottom of the list.

Be particularly alert for instances of items that are supposed to sort in numeric order but that end up being sorted alphabetically. In a numerically sorted list, 9 should come before 10. However, if the list is sorting alphabetically, 10 will come before 9. This is particularly a risk when you have data

2. http://getsatisfaction.com/blundermove/topics/application_closes

that is not, strictly speaking, a pure number. For example, in a list of IP addresses, you might expect it to sort 10.5.4.1, 10.5.4.2, 10.5.4.10. However, if the software is sorting alphabetically, this list will come out as 10.5.4.1, 10.5.4.10, 10.5.4.2.

It turns out that the Turkish *i* also provides an interesting case for positions and sorting. In the Turkish alphabet, there are two variations on the letter *i*, one with a dot and one without.

Files and Storage

Systems often make assumptions about where to find files or data. It might be the location of a file on the file system or the identifier for a hard drive volume. If you can change where a dependent resource lives, you've found a variable.

Changing up the locations can yield interesting surprises. If you move a resource in a distributed system to a location behind a firewall, you might find that the system can no longer access it. Installers often don't handle custom drive locations well. Or if the installer handles a custom location fine, the uninstaller might not. Any number of installers might fail when you attempt to install to any drive other than the default.

Geographic Locations

Software often has logic related to geographic locations: time zones, postal addresses and ZIP codes, elevations, and so on. If you vary addresses or geocodes, you may find that the system can only handle certain locations.

A decade ago, when mapping programs and routing were in their infancies, one routing algorithm worked fine as long as you specified two locations close together. One tester decided to try to plot the route from his work address to the city where he was born, which was about a thousand miles away. It was the first time anyone had attempted such a long route, and the system was completely unable to handle it.

In another case, users accessing a system in different time zones resulted in time stamps showing that a record was edited before it was created. Whenever you can change the physical location of a user or address, choose something far from the default location.

If you tend to use addresses like "123 Main Street" (or whatever equivalent is appropriate for your geographic area), you can increase the power of your exploration by introducing randomness into the location data you use. Use

Google Maps or MapCrunch to select a random location on the globe,[3] or use a test data generator service or tool like fakenamegenerator.com to generate random names and addresses.[4]

Formats

Any number of things have a defined format: dates, mailing addresses, file paths, URLs, the contents of a file, messages, and the list goes on. Any time something can look different and yet retain the same meaning, you have found a format that you can vary.

In some cases there are multiple valid formats:

- In the United States, phone numbers might be represented as either "(866) 867-5309" or "800-867-5309" or even "866.867.5309." They might have the international dialing code as well, like "+1 (866) 867-5309." In the United Kingdom, phone numbers can vary in length.
- Postal code formats vary by country. In the United States, postal codes are either a five-digit number or a nine-digit number represented with a hyphen between the first five and the last four numbers (such as 90051-0345). In Canada, postal codes are six characters and contain letters (such as "M4B 1B4").
- Dates are expressed as month/day/year in the United States (for example, "12/31/2012"), but in Europe they're day/month/year (for example, "31/12/2012.")
- Email addresses can look like "bob@example.com" or like "Bob Smith <bob@example.com>."
- IP addresses can be expressed in v4 format (like "127.0.0.1") or in v6 format (like "::1").
- Pictures might be in .png, .jpg, or .gif format. For that matter, you might try substituting .pdf or .eps files if your system handles images.

Invalid formats are also always fun. Violate domain-specific rules for specific kinds of data: a negative age, an IP address like "999.999.999.999," or a date of February 31.

If your system parses files, you can violate format expectations by corrupting the contents of the file. Throwing random garbage into the file is one way to do it. Another possibility is to create a file with nothing in it. Or you could

3. http://maps.google.com or http://mapcrunch.com, respectively.
4. http://www.fakenamegenerator.com

create a file that is mostly right but is missing some key piece. For example, if your system is parsing an XML file, omit a necessary node or make the XML invalid by removing a closing tag. All these variations are usually interesting.

Size

Obviously files have sizes. If your software imports files, try exploring it with empty or with massively huge files. Similarly, if your software manipulates data in a database, try it with a large database as well as with the more typical near-empty test databases.

Pictures have height and width, and sometimes systems scale or crop images to fit a predefined size. Explore with images of different dimensions as well as different file sizes.

Size matters with hardware configurations too. Installers often include logic to detect the size of a disk drive. Programs often check how much memory is available. One software package I tested misbehaved if the computer was configured with too much memory. Since it is difficult to imagine a system that has too much memory, this might be difficult to believe, but the software had a bug that only showed up when the computer was configured with extra memory.

Software often enforces boundary conditions for sizes around the powers of 2. For example, a common length limit for text fields is 256 characters (2^8). As you are exploring sizes, explore around the powers of 2.

Depth

Anything with a hierarchy has depth. XML data elements can be nested deeply. Files can live deep in the file system. Even equations can have a level of depth with nested parentheses. Exploring by spelunking varying levels of depth often yields interesting surprises and sometimes errors.

An HTML parser I once tested appeared to handle nested tables fine, but the time to parse the file increased exponentially with each level of nesting. The difference wasn't noticeable until I created a file with four levels of nesting—that took a minute or two to open. Encouraged, I tried five levels of nesting. Forty-five minutes later the program was still struggling to parse the file.

The version of Excel that I use has a bug in its floating-point calculations that only manifests with three or more levels of depth of parentheses. Consider the following equation:

```
=1-(100*(1-0.99))
```

It produces the correct result, 0, in my version of Excel. However, when I add one more level of parentheses I get a very different result. This equation looks like it should produce the same result as the equation above:

```
=(1-(100*(1-0.99)))
```

It does not. It yields the result -8.88178E-16.

Timing, Frequency, and Duration

Timing and user actions are always variables. The result of varying timing and sequences of actions can include timeouts, interruptions, and error conditions. Remember the Therac-25 case: at least one of the malfunctions resulted from the operator entering and editing the treatment plan in under eight seconds.

Frequency is an aspect of timing. Try doing actions often and repeatedly. For example, if you refresh Twitter too often you'll hit the requests ceiling and the Twitter server will return an error code. This is expected behavior, but if you have an application dependent on Twitter, it may not handle the error code correctly.

Duration is another aspect of timing. Try varying the duration of your actions, like keeping files or windows open for long periods. One of my favorite tests is to bring up an application and then leave it running overnight. For most apps there is no issue, but some encounter problems. One application I tested had a slow memory leak, so leaving it open overnight meant a crash notification was waiting for me in the morning. Another application I tested had a timeout that terminated sessions after a period of inactivity. Leaving unsaved changes on the screen for ten minutes or more resulted in the loss of whatever changes you were making.

You can use state models to find additional ways to vary timing. Chapter 8, *Discover States and Transitions*, on page 75, discusses how to analyze states, events, and transitions in detail.

Input and Navigation

Even just the way you enter data or manipulate the software can be a variable. If you are exploring a GUI, you might type data in, copy and paste, or drag and drop. You might think that shouldn't make a difference, but it does. Sometimes input validation rules are triggered by one input method but not another. So try the heuristic Violate Data Format Rules with different input methods.

Similarly, the way you navigate a GUI can make a big difference. You might use shortcut keys or the mouse. For example, you might exit a window by clicking a button or by using a shortcut key. Just as with the input method you use, the style of navigation you use can make a big difference in how the software behaves.

4.4 Variables! They're Everywhere!

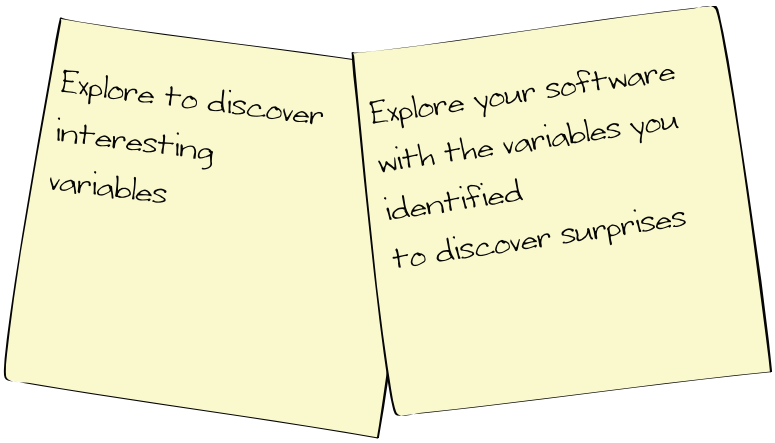
When you start looking for variables, you inevitably notice that they're everywhere. If you already felt as though you had too much to explore and too little time to explore it in, this may feel a bit overwhelming.

One programmer who saw me explaining the concepts in this chapter later told me that he found the talk thoroughly depressing. He said he was overcome by a sense of futility. He realized that no matter how much he explored, there would always be more he could explore.

If you've reached this point in the chapter and are feeling as though exploring is an impossible, never-ending task, take heart. Recognizing that there is always more to test is good news, not bad. It means that you're avoiding the pitfall of complacency. You are becoming a more effective explorer, finding new paths to cover. (Also, you might want to peek ahead to Chapter 13, *Integrate Exploration Throughout*, on page 133, to reassure yourself that there are actually good ways to limit your explorations.)

4.5 Practice Sessions

The best way to become adept at recognizing and manipulating subtle variables is to practice doing it. So that's what your practice charters are about.



Remember when you're executing your charters to look for the following:

- Things you can count and apply the Zero, One, Many heuristic to
- Things you can select and apply the Some, None, All heuristic to
- Things where position matters and apply the Beginning, Middle, End heuristic to
- Things you can move to a different location
- Things with different formats that you can try the Violate Data Format Rules heuristic on
- Things that live in a hierarchy or structure where you can vary the level and nest them deeply
- Hidden settings you can manipulate
- Opportunities to change timing by increasing frequency or extending wait times