

CHAPTER 2

The Case for Manual Testing



“There are two ways to write error-free programs; only the third one works.”

—Alan J. Perlis

The Origin of Software Bugs

The origin of software bugs begins with the very origin of software development itself. It’s clearly not the case that we started out with perfect software and invented ways to screw it up.¹ Indeed, the term *bug* has been in common use within software development from the inception of the discipline² and is a nomenclature that is used today in every office, garage, dorm room, data center, laboratory, bedroom, cafe, and every other place where software is developed. The first software had bugs, the latest software has bugs, and so have all the bits and bytes in between. Software is not, and likely never will be, bug free.

It’s interesting to note that Hopper’s moth (see the second footnote) was not a bug actually created by a programmer. Instead, it was an operational hazard that the developers didn’t consider in their design. As we shall see in later chapters, a developer’s failure to understand, predict, and test potential operational environments continues to be a main source of software failure. Unfortunately, the answer is more complex than closing the windows to keep out the moths. But let’s not get ahead of ourselves. We’ll talk about programmer-created bugs and bugs that creep in through the operational environment throughout this book.

¹ My late mentor Harlan Mills had an interesting take on this: “The only way for errors to occur in a program is by being put there by the author. No other mechanisms are known. Programs can’t acquire bugs by sitting around with other buggy programs.”

² According to Wikipedia, the creation of the term *bug* has been erroneously attributed to one of the earliest and most famous of software developers, Grace Hopper, who, while reacting to a moth that had managed to imbed itself into a relay, called it a bug. Read http://en.wikipedia.org/wiki/Software_bug for earlier uses of the term in hardware, mechanics, and even by Thomas Edison himself.

Preventing and Detecting Bugs

In light of inevitable failure, it seems appropriate to discuss the various ways to keep bugs out of the software ecosystem so as to minimize failures and produce the best software possible. In fact, there are two major categories of such techniques: bug prevention and bug detection.

Preventing Bugs

Bug-prevention techniques are generally developer-oriented and consist of things such as writing better specs, performing code reviews, running static analysis tools, and performing unit testing (which is often automated). All of these prevention techniques suffer from some fundamental problems that limit their efficacy, as discussed in the following subsections.

The “Developer Makes the Worst Tester” Problem

The idea that developers can find bugs in their own code is suspect. If they are good at finding bugs, shouldn't they have known not to write the bugs in the first place? Developers have blind spots because they approach development from a perspective of *building* the application. This is why most organizations that care about good software hire a second set of eyes to test it. There's simply nothing like a fresh perspective free from development bias to detect defects. And there is no replacement for the tester attitude of *how can I break this* to complement the developer attitude of *how can I build this*.

This is not to say that developers should do no testing at all. Test-driven development or TDD is clearly a task meant as a development exercise, and I am a big believer in unit testing done by the original developer. There are any number of formatting, data-validity, and exception conditions that need to be caught while the software is still in development. For the reasons stated previously, however, a second set of eyes is needed for more subtle problems that otherwise might wait for a user to stumble across.

The “Software at Rest” Problem

Any technique such as code reviews or static analysis that doesn't require the software to actually run, necessarily analyzes the software *at rest*. In general, this means techniques based on analyzing the source code, object code, or the contents of the compiled binary files or assemblies. Unfortunately, many bugs don't surface until the software is running in a real operational environment. This is true of most of the bugs shown previously in Chapter 1, “The Case for Quality Software”: Unless you run the software and provide it with real input, those bugs will remain hidden.

The “No Data” Problem

Software needs input *and* data to execute its myriad code paths. Which code paths actually get executed depends on the inputs applied, the software's internal state (the values stored in internal data structures and variables),

and external influences such as databases and data files. It's often the accumulation of data over time that causes software to fail. This simple fact limits the scope of developer testing, which tends to be short in duration.

Perhaps tools and techniques will one day emerge that enable developers to write code without introducing bugs.³ Certainly it is the case that for narrow classes of bugs such as buffer overflows⁴ developer techniques can and have driven them to near extinction. If this trend continues, the need for a great deal of testing will be negated. But we are a very long way, decades in my mind, from realizing that dream. Until then, we need a second set of eyes, running the software in an environment similar to real usage and using data that is as rich as real user data.

Who provides this second set of eyes? Software testers provide this service, using techniques to detect bugs and then skillfully reporting them so that they get fixed. This is a dynamic process of executing the software in varying environments, with realistic data and with as much input variation as can be managed in the short cycles in which testing occurs. Such is the domain of the software tester.

Detecting Bugs

Testers generally use two forms of dynamic analysis: automated testing (writing test code to test an application) and manual testing (using shipping user interfaces to apply inputs manually).

Automated testing carries both stigma and respect.

The stigma comes from the fact that tests are code, and writing tests means that the tester is necessarily also a developer. Can a developer really be a good tester? Many can, many cannot, but the fact that bugs in test automation are a regular occurrence means that they will spend significant time writing code, debugging it, and rewriting it. Once testing becomes a development project, one must wonder how much time testers are spending thinking about testing the software as opposed to maintaining the test automation. It's not hard to imagine a bias toward the latter.

The respect comes from the fact that automation is cool. One can write a single program that will execute an unlimited number of tests and find tons of bugs. Automated tests can be run and then rerun when the application code has been churned or whenever a regression test is required. Wonderful! Outstanding! How we must worship this automation! If testers

³ In my mind, I picture the ultimate developer bug-finding tool to work on their code as they type. It will work in a way similar to spell checkers for documents. The moment a developer types a bug into the code editor, the errant fragment will be underlined or, perhaps, corrected automatically. The whole point is that we place the *detection* of the bug as close as possible to the *creation* of the bug so that the bug doesn't get into the software at all. The less time a bug lives, the better off we'll all be. But until such technology is perfected, we'll have to keep on testing. We're in this for the long haul!

⁴ Buffer overflows are found by injecting more data into an input field than the underlying code can handle. Their cause and specific ways to find them are explained in *How to Break Software Security* (Addison-Wesley, 2004) on page 41.

are judged based on the number of tests they run, automation will win every time. If they are based on the *quality* of tests they run, it's a different matter altogether.

The kicker is that we've been automating for years, decades even, and we still produce software that readily falls down when it gets on the desktop of a real user. Why? Because automation suffers from many of the same problems that other forms of developer testing suffers from: It's run in a laboratory environment, not a real user environment, and we seldom risk automation working with real customer databases because automation is generally not very reliable. (It is *software* after all.) Imagine automation that adds and deletes records of a database—what customers in their right mind would allow that automation anywhere near their real databases? And there is one Achilles heel of automated testing that no one has ever solved: the oracle problem.

The oracle problem is a nice name for one of the biggest challenges in testing: *How do we know that the software did what it was supposed to do when we ran a given test case?* Did it produce the right output? Did it do so without unwanted side effects? How can we be sure? Is there an oracle we can consult that will tell us—given a user environment, data configuration, and input sequence—that the software performed exactly as it was designed to do? Given the reality of imperfect (or nonexistent) specs, this just is not a reality for modern software testers.

Without an oracle, test automation can find only the most egregious of failures: crashes, hangs (maybe), and exceptions. And the fact that automation is itself software often means that the crash is in the test case and not in the software! Subtle/complex failures are missed. One need look no further than the previous chapter to see that many such crucial failures readily slip into released code. Automation is important, but it is not enough, and an overreliance on it can endanger the success of a product in the field.

So where does that leave the tester? If a tester cannot rely on developer bug prevention or automation, where should she place her hope? The only answer can be in manual testing.

Manual Testing

Manual testing is human-present testing. A human tester uses her brain, her fingers, and her wit to create the scenarios that will cause software either to fail or to fulfill its mission. Human-present testing allows the best chance to create realistic user scenarios, using real user data in real user environments and still allowing for the possibility of recognizing both obvious and subtle bugs.

Manual testing is the best choice for finding bugs related to the underlying business logic of an application. Business logic is the code that implements user requirements; in other words, it is the code that customers buy the software for. Business logic is complex and requires a human in the loop to verify that it is correct, a task that automation is too often ill-suited to accomplish.

Perhaps it will be the case that developer-oriented techniques will evolve to the point that a tester is unnecessary. Indeed, this would be a desirable future for software producers and software users alike, but for the foreseeable future, tester-based detection is our best hope at finding the bugs that matter. There is simply too much variation, too many scenarios, and too many possible failures for automation to track it all. It requires a “brain in the loop.” This is the case for this decade, the next decade, and perhaps a few more after that.

I wish it was just that easy, but historically the industry has not been good at manual testing. It’s too slow, too ad hoc, not repeatable, not reproducible, not transferable, and there isn’t enough good advice out there for testers to get good at it. This has created a poor reputation for manual testing as the ugly stepchild of development. It’s unfortunate that this is the case, but such is the hand we are dealt.

It’s time we put the best technology available into the process of manual testing. This is the subject of exploratory testing that this book addresses. I want the industry to get past the idea of ad hoc manual testing and work toward a process for exploratory testing that is more purposeful and prescriptive. It should be a process where manual testing requires careful preparation yet leaves room for intelligent decision making during testing. Manual testing is too important to treat it with any less respect.

We may look to a future in which software just works, but if we achieve that vision, it will be the hard work of the manual testers of this time that makes it possible.

Scripted Manual Testing

Many manual testers are guided by scripts, written in advance, that guide input selection and dictate how the software’s results are to be checked for correctness. Sometimes scripts are specific: Enter this value, press this button, check for that result, and so forth. Such scripts are often documented in spreadsheet tables and require maintenance as features get updated through either new development or bug fixes. The scripts serve a secondary purpose of documenting the actual testing that was performed.

Often, scripted manual testing is too rigid for some applications, or test processes and testers take a less-formal approach. Instead of documenting every input, a script may be written as a general scenario that gives some flexibility to the testers while they are running the test. At Microsoft, the folks who manually test Xbox games often do this. So an input would be “interact with the mage,” without specifying exactly the type of interaction they must perform. Thus it is possible that scripted testing can be as rigid or as flexible as necessary, but for the flexibility to work, testers are going to need very specific advice for how to handle choice and uncertainty, and this is more the domain of exploratory testing.

In this book, we are only interested in the flexible type of scripted testing.

Exploratory Testing

When the scripts are removed entirely (or as we shall see in later chapters, their rigidity relaxed), the process is called *exploratory testing*. Testers may interact with the application in whatever way they want and use the information the application provides to react, change course, and generally explore the application's functionality without restraint. It may seem ad hoc to some, but in the hands of a skilled and experienced exploratory tester, this technique can prove powerful. Advocates argue that exploratory testing allows the full power of the human brain to be brought to bear on finding bugs and verifying functionality without preconceived restrictions.

Testers using exploratory methods are also not without a documentation trail. Test results, test cases, and test documentation are generated as tests are being performed instead of being documented ahead of time in a test plan. Screen capture and keystroke recording tools are ideal for recording the result of exploratory testing. Just because it's manual testing doesn't mean we can't employ automation tools as aids to the process. Indeed, even those who "handcraft" furniture do so with the assistance of power tools. Handcrafting test cases should be no different. Manual testers who use debug builds, debuggers, proxies, and other types of analysis tools are still doing manual testing; they are just being practical about it.

Exploratory testing is especially suited to modern web application development using agile methods.⁵ Development cycles are short, leaving little time for formal script writing and maintenance. Features often evolve quickly, so minimizing dependent artifacts (like pre-prepared test cases) is a desirable attribute. If the test case has a good chance of becoming irrelevant, why write it in the first place? Are you not setting yourself up for spending more time maintaining test cases than actually *doing* testing?

The drawback to exploratory testing is that testers risk wasting a great deal of time wandering around an application looking for things to test and trying to find bugs. The lack of preparation, structure, and guidance can lead to many unproductive hours and retesting the same functionality over and over, particularly when multiple testers or test teams are involved. Without documentation, how do testers ensure they are getting good coverage?

This is where guidance comes into play. Exploratory testing without good guidance is like wandering around a city looking for cool tourist attractions. It helps to have a guide and to understand something about your destination (in our case, software) that can help your exploration to be more methodical. Looking for beaches in London is a waste of time. Looking for medieval architecture in Florida is equally so. Surely *what* you are testing is just as important to your strategy as how you test it.

⁵ The number of proponents of exploratory testing is large enough now that its case no longer needs to be argued, particularly among the agile development community. However, I argue it here anyway to help those testers who still have to convince their management.

There are two types of guidance for exploratory testers to help in the decision-making process: *exploratory testing in the small*, which aids in local decision making while running tests; and *exploratory testing in the large*, which helps testers design overall test plans and strategies. Both are summarized here and covered in detail in Chapter 3, “Exploratory Testing in the Small,” and Chapter 4, “Exploratory Testing in the Large.” Finally, a third class of exploratory testing that combines elements of exploration with scripted manual testing is discussed in Chapter 5, “Hybrid Exploratory Testing Techniques.”

Exploratory Testing in the Small

Much of what a manual tester does is about variation. Testers must choose which inputs to apply, what pages or screens to visit, which menu items to select, and the exact values to type into each input field they see. There are literally hundreds of such decisions to make with every test case we run.

Exploratory testing can help a tester make these decisions. And when a tester uses exploratory testing strategy to answer these sorts of questions, I call this *exploratory testing in the small* because the scope of the decision is small. A tester is looking at a certain web page or dialog box or method and needs focused advice about what to do for that specific situation. This is necessarily a localized decision-making process that testers will perform dozens of times in a single test case and hundreds of times over the course of a day of testing.

The problem is that many testers don’t know what to do in the variety of “small” situations that they encounter. Which value do you enter into a text box that accepts integers? Is the value 4 better (meaning more likely to find a bug or force a specific output) than the value 400? Is there anything special about 0 or about negative numbers? What illegal values might you try? If you know something about the application—for example, that it is written in C++ or that it is connected to a database—does that change the values you might try? What, indeed, is the sum total of exploratory testing wisdom that we can use to help us make the right small decisions as we test?

Chapter 3 is devoted to passing along some of this wisdom. I’ll be the first to admit, that most of it is not mine. I’ve been lucky enough to work around some of the best software testers to grace this planet. From IBM to Ericsson to Microsoft, Adobe, Google, Cisco, and many more names far less recognizable, I’ve gathered what I think is a large portion of this advice and I reproduce it here. Much of this information was embodied in *How to Break Software*, and so readers of that book can consider this an update to the body of knowledge that was published there. But as the attitude of that book was about finding bugs, the purpose of this book is much broader. Here we are interested in more than finding bugs. We want to force software to exhibit its capabilities and gain coverage of the application’s features, interfaces, and code and find ways to put it through its paces to determine its readiness for release.

Exploratory Testing in the Large

There is more to testing, however, than making all the small decisions correctly. In fact, it is possible to nail all the small decisions and still not have an overall set of tests that confirm (or reject) release readiness. The sum total of all the test cases is definitely more than the individual parts. Test cases are interrelated, and each test case should add to the others and make the entire set of test cases better in some substantive, measurable (or at least arguable) way.

This points to the need for a strategy that guides test case design and exploration. Which features should a single test case visit? Are there certain features or functions that must be tested together? Which feature should be used first, and how do we decide which subsequent features to test? If there are multiple testers on a project, how can we make sure their strategies complement each other and they don't end up testing the same things? How does an exploratory tester make these larger scope decisions about overall test cases and testing strategy?

I call this *exploratory testing in the large* because the scope of the decisions to be made encompasses the software as a whole instead of a single screen or dialog. The decisions made should guide how an application is explored more than how a specific feature is tested.

In Chapter 4, I use a tourism metaphor to guide exploratory testing in the large. Think about it this way: As a tourist visiting a new city, you will use *in-the-large* advice to choose which restaurant to visit, but you will use *in-the-small* advice to choose what meal and drink to order. *In-the-large* advice will help plan your entire day and advise you on how to plan your entire stay, the landmarks you visit, the shows you see, and the restaurants at which you dine. *In-the-small* advice will help you navigate each of these events and plan the subtle details that a larger plan will always leave out. By perfecting the combination of the two, you've entered the world of an expert exploratory software tester.

Combining Exploration and Scripting

It isn't necessary to view exploratory testing as a strict alternative to script-based manual testing. In fact, the two can co-exist quite nicely. Having formal scripts can provide a structure to frame exploration, and exploratory methods can add an element of variation to scripts that can amplify their effectiveness. The expression *opposites attract* is relevant in the sense that because formal scripts and exploratory methods are at opposite extremes of the manual testing spectrum, they actually have a lot to offer each other. If used correctly, each can overcome the other's weaknesses, and a tester can end up in the happy midpoint of a very effective combination of techniques.

The best way that I have found to combine the two techniques is to start with formal scripts and use exploratory techniques to inject variation into

them. This way, a single script may end up being translated into any number of actual exploratory test cases.

Traditional script-based testing usually involves a starting point of user stories or documented end-to-end scenarios that we expect our eventual users to perform. These scenarios can come from user research, data from prior versions of the application, and so forth, and are used as scripts to test the software. The added element of exploratory testing to traditional scenario testing widens the scope of the script to inject variation, investigation, and optional user paths.

An exploratory tester who uses a scenario as a guide will often pursue interesting alternative inputs or pursue some potential side effect that is not included in the script. However, the ultimate goal is to complete the scenario so that these testing detours always end up back on the main user path documented in the script. The detours from the script can be chosen based on structured ways of modifying specific steps in the script or by exploratory excursions off the script and then back again. Chapter 5 is dedicated entirely to script-based exploratory testing because it is one of the key tools in the manual tester's arsenal of techniques.

The techniques in Chapters 3 through 5 have been applied in a number of case studies and trials throughout Microsoft, and the results are presented in Chapter 6, "Exploratory Testing in Practice," as experience reports written by the testers and test leads involved in these projects. Chapter 6 examines how the exploratory testing techniques were applied to several different classes of software from operating system components to mobile applications to more traditional desktop and web software. Also, special tours written specifically for a particular project are described by their authors.

The remainder of the book highlights essays on, respectively, building a testing career and the future of testing, followed by past and current essays and papers while I was a professor at Florida Tech and an architect at Microsoft. Since I have now left Microsoft, this book may be the only place that the latter material can be found.

Conclusion

The world of manual exploratory testing is one of the most challenging and satisfying jobs in the IT industry. When done properly, exploratory testing is a strategic challenge and a match of wits between the manual tester and an application to discover hidden bugs, usability issues, security concerns, and so forth. For far too long, such exploration has been done without good guidance and has been the realm of experts who have learned their craft over many years and decades. This book contains much of that experience and wisdom in the hopes that many new experts emerge quickly, allowing

higher-quality testing and thus higher-quality applications to enter the technology ecosystem.

We need the human mind to be present when software is tested. The information in this book is aimed at focusing the human mind so that testing is as thorough and complete as possible.

Exercises

1. Why can't we just build software to test software? Why isn't automation the answer to the software-testing problem?
2. What type of code is automation good at testing? What type of code requires manual testing? Try to form a theory to explain why.
3. What types of bugs is automation good at detecting? What types of bugs is automation bad at detecting? Give examples.