A Context-Driven Approach to Automation in Testing

By James Bach and Michael Bolton February, 2016 (v1.05)

There are many wonderful ways tools can be used to help software testing. Yet, all across industry, tools are poorly applied, which adds terrible waste, confusion, and pain to what is already a hard problem. Why is this so? What can be done? We think the basic problem is a shallow, narrow, and ritualistic approach to tool use. This is encouraged by the pandemic, rarely examined, and absolutely false belief that testing is a mechanical, repetitive process. Good testing, like programming, is instead a challenging intellectual process. Tool use in testing must therefore be mediated by people who understand the complexities of tools and of tests. This is as true for testing as for development, or indeed as it is for any skilled occupation from carpentry to medicine.

Thank you to our reviewers: Tim Western, Alan Page, Keith Klain, Ben Simo, Paul Holland, Alan Richardson, Christin Wiedemann, Noah Sussman, and Joseph Quaratella.

James Bach, james@satisfice.com, http://www.satisfice.com, @jamesmarcusbach
Michael Bolton, michael@developsense.com, http://www.developsense.com, @michaelbolton
Copyright 2016, Satisfice, Inc., all rights reserved

A Context-Driven Approach to Automation in Testing

By James Bach and Michael Bolton February, 2016 (v1.05)

Table of Contents

Robots! Help! 2
The Trouble with "Automation" 3
First: Call them tools (not "test automation"). 4
Second: Think of testing as much more than output checking. 6
Distinguish between checking and testing. 7
Checking is important. 8
Third: Explore the many ways to use tools! 8
Let your context drive your tooling. 9
How specifically does context drive tooling? 10
Invest in tools that give you more freedom in more situations. 12
Invest in testability. 14
Let's see tool-supported testing in action! 15
CASE #1: Tool use without checking. 15
CASE #2: Tool-support via patterned data generation for better coverage and a powerful oracle 17
CASE #3: Automated checking. 20
Why is automating interactions through a GUI so difficult? 23
Automating actions is a tactic. It should not be a ritual 25

In this white paper, we offer a vision of test automation that puts the tester at the center of testing, while promoting a way of thinking that celebrates the many things tools can do for us. We embrace tools without abdicating our responsibility as technical people to run the show.

Tools can be powerful, and we are going to say encouraging and helpful things about them. But automation can also be treacherous—not least because the label "automation" refers to a mess of different things. So, we must begin with a sober look at some basic misconceptions that add terrible waste, confusion, and pain to what is already difficult even in the best of times. If you need good testing, then good tool support will be part of the picture, and that means you must learn why we go wrong with tools.

Robots! Help!

We can summarize the dominant view of test automation as "*automate testing by automating the user.*" We are not claiming that people literally say this, merely that they try to do it. We see at least three big problems here that trivialize testing:

1. The word "automation" is misleading. We cannot automate *users*. We automate *some* actions they perform, but users do so much more than that.
2. Output checking can be automated, but *testers* do so much more than that.
3. Automated output checking is interesting, but *tools* do so much more than that.

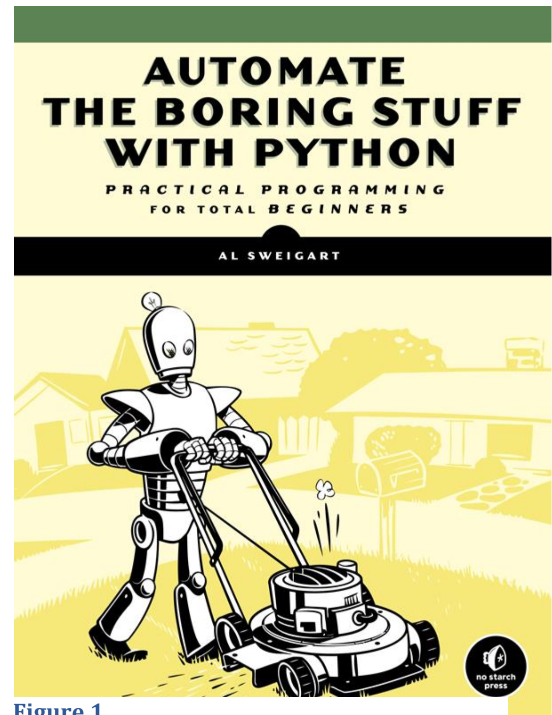

Figure 1

Automation comes with a tasty and digestible story: replace messy, complex humanity with reliable, fast, efficient robots! Consider Figure 1. It perfectly summarizes the impressive vision: "Automate the Boring Stuff." Okay. What does the picture show us?

It shows us a machine that is intended to function as a human. The robot is constructed as a humanoid. It is using a tool normally operated by humans, in exactly the way that humans would operate it, rather than through an interface more suited to robots. There is no depiction of the process of programming the robot or controlling it, or correcting it when it errs. There are no broken down robots in the background. The human role in this scene is not depicted. No human appears even in the background. The message is: robots replace humans in uninteresting tasks without changing the nature of the process, and without any trace of human presence, guidance, or purpose. Is that what automation is? Is that how it works? No!

Of course it is a light-hearted cartoon, not to be taken seriously. The problem is, in our travels all over the industry, we see clients thinking about real testing, real automation, and real people in just this cartoonish way. The trouble that comes from that is serious.

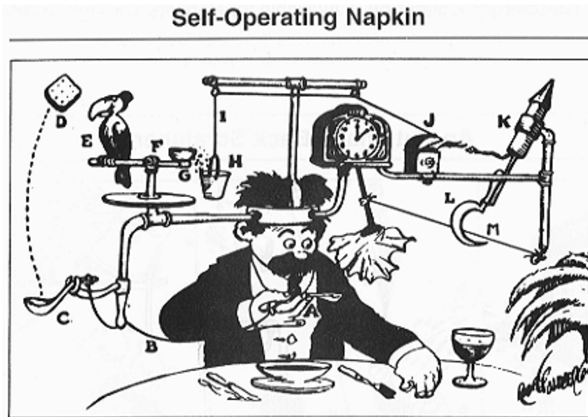

Figure 2: Cartoon by Rube Goldberg

How serious? In the experience of the authors, observing projects going back to the 80's, we find that it is normal for large scale automation efforts to lavish the bulk of their budgets in the detection of trivial and obvious GUI-level bugs, drawing much needed time and effort away from the hunt for serious but subtle problems—what we call *deep* bugs. Furthermore, the typical automation approach has the character of a Rube Goldberg machine—swimming in dependencies and almost comically prone to breakdown.[1] This sort of automation becomes almost like a new stakeholder on the project; as with some obsessive-compulsive "high maintenance" cleaning lady who won't even enter the house until it is already spotless. We believe the effort typically invested in automation would in most cases be better invested directly into humans interacting with the product in complex and sophisticated ways (which also finds the shallow bugs) and into less expensive supporting tools that help testers test better.

No one can deny that automation tool sales demos are impressive. What we deny is that people agree on what "automation" means, what it should be, and that those sales demos translate into practical value on ordinary projects.

The Trouble with "Automation"

The trouble with "test automation" starts with the words themselves. Testing is a part of the creative and critical work that happens in the design studio, but "automation" encourages people to think of mechanizable assembly-line work done on the factory floor.

The term "test automation" is also ambiguous. It is common to hear someone say a sentence like "run the test automation," which refers specifically to tools. A sentence like "test automation is worth doing" refers not only to tools but also to the enterprise of creating, maintaining, testing, and operating those tools. In the first sense, test automation is not human at all. It's incredibly fast and inexpensive, too, since you don't pay the computer. In the second sense, test automation is a skilled activity performed by humans who write and operate software over hours, days, or weeks—and those people must be paid for their time.

---

[1] In one case, James was called in to help a project that had "more than 3,000" automated scripts, developed over a nine-month period. James asked to see them executed, whereupon it was revealed that they *all* had been broken by a recent update to their expensive commercial test tool and ongoing updates to their own product.

We observe that in common parlance, the driving tactic of "test automation" is to script ordinary, rote actions of a user of the product—and a rather complacent, unimaginative user, at that— then get the machinery to punch the keys at dazzling speed, and then check to see whether specified actions and inputs produce specified outputs. From there, it's a small step to start thinking of "test automation" as a sort of tester in its own right. But even a minimally skilled human tester does far more than blindly adhere to prescribed actions, and observes far more than the output of some function. Humans have complicated lives, agendas, talents, ideas, and problems. Although certain user and tester actions can be *simulated*, users and testers themselves cannot be *replicated* in software. Failure to understand this simple truth will trivialize testing, and will allow many bugs to escape our notice.

"We define a tool as any human contrivance that aids in fulfilling a human purpose."

How can we think about all this more clearly?

First: Call them tools (not "test automation").

We define a tool as **any human contrivance that aids in fulfilling a human purpose**. A test tool could be software; hardware; a map, document, or artifact; or some other heuristic that aids in fulfilling a testing purpose. We are primarily concerned with software-based tools, here.

The term "test tool" connects us to the ordinary, everyday understanding that these contrivances do not work without human guidance; they extend the capabilities of an appropriately skilled human. Moreover, "tool" opens the door to the many ways that tools can lighten burdens and amplify the power of testers.

Meanwhile, the term "test automation" threatens to *dissociate* people from their work. To understand why, you must consider what testing is. To test is to seek the true status of a product, which in complex products is hidden from casual view. Testers do this to discover trouble. A tester, working with limited resources, must sniff out trouble before it's too late. This requires careful attention to subtle clues in the behavior of the product within a rapid and ongoing learning process. Testers engage in sensemaking, critical thinking, and experimentation, none of which can be done by mechanical means. Yet, in our long experience visiting hundreds of companies and teams, we find managers and technocrats who speak of testing routinely ignore these intellectual processes. We have tried reminding them—and our own colleagues, at times—of these crucial elements that cannot be encoded into test cases or test software. "Oh we agree," they might say, but then lapse back into speaking exactly as if the essence of testing is somehow expressed in their "test automation." After years of this struggle, we conclude that the term itself is a sort of narcotic.

Computer software is comprised strictly of explicitly encoded patterns. Any valuable or important pattern of behavior will not happen unless it is expressed in code. This is obvious. What is not so obvious is that much of what informs a human tester's behavior is tacit knowledge[2]. (Whereas, explicit knowledge is any knowledge that is represented as a string of bits, tacit knowledge is that which is not or cannot be so represented.)

When a human tester interacts with a product, he spontaneously reacts to an astonishing variety of surprising and erroneous events without ever having been consciously aware of an expectation about them. If, for instance, a window turns purple for a moment, or an extra line appears, or a process takes a little longer to complete one out of ten times, he almost effortlessly notices and reacts. But when this tester tells the story of this test, perhaps by writing down its steps and expected results, only a small part of all those real expectations are expressed. No tester will encode an *unconscious* expectation or *unanticipated* action. Since the testing humans actually do cannot be put into words, it cannot be encoded and therefore cannot be automated. We should not use a term that implies it can be.

"Since the testing humans actually do cannot be put into words, it cannot be encoded and therefore cannot be automated."

Everyone knows *programming* cannot be automated. Although many early programming languages were called "autocodes" and early compilers were called "autocoders," that way of speaking peaked around 1965[3]. The term "compiler" became far more popular. In other words, when software started coding, *they changed the name of that activity to compiling, assembling, or interpreting*. That way the programmer is someone who always sits on top of all the technology and no manager is saying "when can we automate all this programming?"

To produce high-quality products and services, we need *skilled people applying appropriate tools to fulfill the mission of testing.* The common terms "manual testers" or "automated testers" to distinguish testers are misleading, because all competent testers use tools. Programmers and researchers use tools, too, but no one speaks of "automated programming" or "automated research." No manager who calls for automated testing aspires to automate his management. The only reason people consider it interesting to automate testing is that they honestly believe testing requires no skill or judgment.

Since all testers use tools, we suggest a more interesting distinction is that *some* testers also *make* tools—writing code and creating utilities and instruments that aid in testing. We suggest calling such technical testers "toolsmiths." Although toolsmiths and their tools help to extend, accelerate,

---

[2] An excellent source for learning about this is *Tacit and Explicit Knowledge* by Harry Collins. We call Harry the "sociologist for testers" because his studies of scientists at work apply perfectly to the world of testing, too.
[3] According to Google Ngram Viewer

and intensify certain activities within testing, they do not automate testing itself. Therefore, from this point on, we shall try to avoid using the term "test automation."

Second: Think of testing as much more than output checking.

We say that testing is *evaluating a product by learning about it through exploration and experimentation, which includes to some degree: questioning, study, modeling, observation and inference, etc.* [4]

We choose our words carefully. Testing is necessarily a human process. Only humans can learn. Only humans can determine value. Value is a social judgment, and different people value things differently. Technologists may believe that they can automate the evaluation of requirements by encoding them into a script, but the evaluation is provisional and incomplete until it has been reviewed by a human. There are nearly always circumstances in which a manager will say "the tool is reporting a bug, but it is really not a problem in this case."

Good Checking is a Subset of Testing

Checking is not the same as testing in the way that biting is not the same as eating; tires are not the same as cars; and spell checking is not the same as editing. Good checking is always a product of and embedded in a test process. Testing gives checking its value and meaning.

Exploration is central to our definition of testing because we don't know where the bugs are before we find them. Indeed, with any new product we must discover where to look for problems, and there are too many places to look for us to check them all. We don't even know for sure what counts as a bug; that is a judgment that drifts and shifts over the course of a project. We emphasize experimentation because good tests are literally experiments in the scientific sense of the word. At least 300 years before anyone ever wondered what software could or would do, "natural philosophers" were systematically testing nature via their experiments.[5] What scientists mean by experiment is precisely what we mean by test. Testing is necessarily a process of incremental, speculative, self-directed search.

Finally, the "etc." at the end is a signal that testing incorporates many other analysis-related activities and disciplines. Activities

---

[4] See http://www.satisfice.com/blog/archives/856

[5] "...what these several degrees are I have not yet experimentally verified; but it is a notion, which if fully prosecuted as it ought to be, will mightily assist the astronomer to reduce all the Celestial motions to a certain rule, which I doubt will never be done true without it..." Robert Hooke, 1674, An Attempt to Prove the Motion of the Earth by Observations, (http://bit.ly/1MDwhBI)

that aren't themselves testing, such as studying a specification, become testing when done *for the purposes of testing*.

Let's break down testing further. What do we specifically do when we test? Testing is a performance that involves several kinds of ongoing, parallel activities:

- we *design* our testing by learning and modelling the product, determining test conditions to cover, generating specific test data, identifying and developing oracles (i.e. the means to recognize problems when we encounter them), and establishing procedures to explore and experiment.
- we *interact with the product* by configuring, operating and observing it.
- we *evaluate* the product by using appropriate oracles to detect inconsistencies between the product and qualities that we might consider ideal.
- we *record and report* the testing work that has been done.
- we *manage* the testing work, which includes understanding the current status of testing, analyzing product risk, scoping and assigning testing tasks.

All of these activities can be helped with tools.

Distinguish between checking and testing.

We find it necessary to distinguish between checking and testing. Checking is **the process of making evaluations by applying algorithmic decision rules to specific observations of a product**. This is different from the rest of testing in one vital way: *it can be completely automated*. Checking is an appropriate place to use that word "automation."

In testing, we design and perform experiments that help us develop our understanding of the status of the product. This understanding is an interpretation; an assessment. *But it is not a fact.* Simple facts are arguably "verifiable," but quality is never a simple fact. Quality is a working hypothesis. When you exercise software and fail to spot a specific problem, you have not proven or demonstrated that "it works." All you know is that you haven't yet recognized a failure. All you have demonstrated is that the product *can* work. The product may have failed in a subtle way you did not or cannot yet detect., Maybe it works fine now, but won't work ten minutes from now. So does it really, truly, deeply work? *No output check can tell you that. No collection of output checks can tell you that.*

Indeed, any advertiser, late-night TV pitchman, or stage magician can show you that something *appears* to work. Our job as testers is not to obey the ad, swallow the pitch, or believe the trick. Our job is to figure out what the ad leaves out, where the product doesn't meet the claims, or how the magician might be fooling us. Although routine output checking is part of our work, we continually re-focus on non-routine, novel observations. Our attitude must be one of seeking to find trouble, not verifying the absence of trouble—otherwise we will test in shallow ways and blind ourselves to the true nature of the product.

Evaluating quality is a task that requires skillful, complex, non-algorithmic investigation and judgment. That task can be supported and accelerated by tools, but it cannot be performed by the tools themselves.

Checking is important.

Good checking is a subset of testing. Checking is not the same as testing in the way that biting is not the same as eating; tires are not the same as cars; and spell checking is not the same as editing. Good checking is always a product of— and embedded in— the processes of designing, implementing, and interpreting those checks, which are *human* activities; which constitute testing. Testing gives checking its value and meaning. Whereas, checking keeps testing grounded.

Automated checking is a *tactic* of testing, and can have considerable value. Programmers who adopt automated checks into their coding practices can provide themselves with fast, inexpensive feedback. Checking through an API beneath the GUI level can be particularly useful. In designing such low-level checks, programmers and testers can profitably work together.

We are more doubtful of automated checking at the GUI level. GUIs are notoriously fussy. Because non-technical people can see them and discuss them, GUIs may change much more capriciously than the underlying interfaces that only programmers see. This can lead to a large, expensive maintenance effort just to keep the simple checks running. Moreover, GUIs are designed to feel natural and comfortable for people, not for other software. You may need a skilled full-time programmer to maintain all the code necessary to attempt to simulate a speedy but unskilled human tester. That is probably not a money-saving proposition.

Third: Explore the many ways to use tools!

The skill set and the mindset of the individual tester are central to the responsible use of tools. When we say this, however, some people seem to hear us saying that tools are not important, or that context-driven testers hate tools. *Nothing could be farther from the truth.*

Let's catalog some of the many ways tools help us in testing:

- **In design; we use tools to help us**
  - produce test data (tools like spreadsheets; state-model generators; Monte Carlo simulations; random number generators)
  - obfuscate or cleanse production data for privacy reasons (data shufflers; name replacers)
  - generate interesting combinations of parameters (all-pairs or combinatorial data generators)
  - generate flows through the product that cover specific conditions (state-model or flow-model path generators)

- **In product interaction, we use tools to help us**
  - set up and configure the product or test environments (like continuous deployment tools; virtualization tools; or system cloning tools)
  - submitting and timing transactions; perhaps for a long time; at high volume; under stress (profiling and benchmarking tools)
  - encode procedures like operating the product and comparing its outputs to calculated results (this is automated checking).
  - simulate software or hardware that has not been developed yet; or that we do not have immediately available to us (mocking or stubbing tools)
  - probe the internal state of the system and analyze traffic within it as testing is being performed (instrumentation; log analysis; file or process monitors; debugging tools)
- **In evaluation, we use tools to help us**
  - sort, filter, and parse output logs (text editors; spreadsheets; regular expressions)
  - visualize output for comparative analysis (diffing, charting and graphing tools, conditional output formatting)
  - develop, adapt and apply oracles that help us recognize potential problems (source file or output comparison tools; parallel or comparable algorithms; internal consistency checks within the application; statistical analysis tools)
- **In recording and reporting, we use tools to help us**
  - record our activities and document our procedures (note-taking tools; video-recording tools; built-in logging; word processing tools; user interaction recording tools)
  - prepare reports for our clients (mind maps; word processors; spreadsheets; presentation software)
- **In managing the testing work, we use tools to help us**
  - map out our strategies (mind maps, outline processors, word processors)
  - identify what has and has not been covered by testing (coverage tools; profilers; log file analysis)
  - preserve information about our products, and to aid other people in future support and development (wikis; knowledge bases; file servers)

And this is an *incomplete* list of the ways in which we use tools to help us. Moreover, we use tools to help us produce the tools that we use.

You have probably noticed how we repeatedly said “We use tools to help us...” We have chosen these words deliberately to emphasize once again that *tools don’t do testing work; tools help testers to do testing work*. In conversation about testing, tools may be important, but the center of testing must be the skill set and the mindset of the individual tester.

Let your context drive your tooling.

By “context”, we mean **the set of factors that should affect the decisions of a responsible tester**. Generally speaking, a craftsman who has both the skills and intent to select and apply the appropriate tools and methods for any given context can be called *context-driven*. More specifically the Context-Driven school of software testing is a paradigm of testing based on the following principles:

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project’s context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn’t solved, the product doesn’t work.
6. Good software testing is a challenging intellectual process.
7. Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.

These principles were written by Cem Kaner, James Bach, and Bret Pettichord, and first published in their book *Lessons Learned in Software Testing: A Context-Driven Approach*, which is the seminal book on Context-Driven thinking.

Note that if you are working in a way that solves the problems that exist in your environment, you may be doing context-*specific* work without necessarily being context-*driven*. To be context-driven you must be ready and able to change the way you work if and when the context changes. That’s why the Context-Driven community focuses on developing skills and sharing experiences across many kinds of projects and technologies. This is why we foster peer conferences dedicated to conversation and debate.

How specifically does context drive tooling?

Context drives tooling through the activity of ongoing problem-solving. We do that by developing in our minds various understandings, including:

- what surrounds us and our place in that world
- our clients and our mission
- other people involved and what they are trying to do
- tools and techniques available to us
- actions we could take and the effects they may have
- the immediate costs (and time) of those actions
- the long term costs of those actions
- the value of learning from trying new things

These understandings may be thought of as spaces that we explore throughout our projects and careers. As we learn and grow, during the course of our projects and careers, we get better at navigating them.

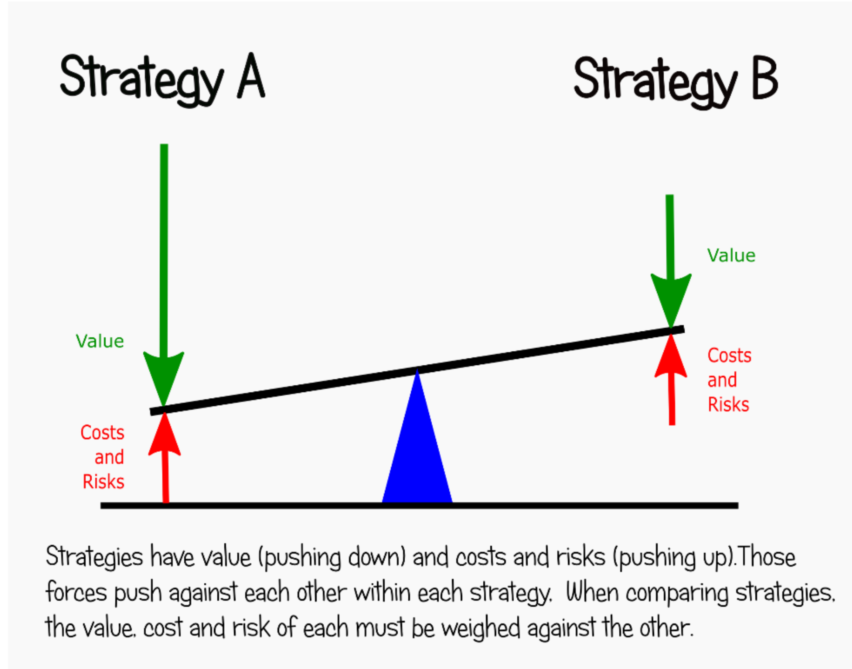

Figure 3

What we do with those understandings ultimately results in mental calculations and decisions along the lines of Figure 3. In context-driven work, our choices are guided not according to a fixed script of "best practices" but rather by dynamically evaluating context and selecting, designing, or adjusting our actions to solve the problems that we encounter. We don't simply look at whether a particular strategy is worth doing in and of itself, such as strategy B in the diagram, where you can see that its value outweighs the risks and the costs. We also compare that to other strategies that might be even better, such as strategy A. Yes, these decisions may be biased, as in Figure 4, perhaps because we unconsciously veer toward things we know and away from potentially wonderful new ideas that we aren't yet comfortable with. But still, we strive to make decisions based on merits rather than following the dictates of fashion or arguments from authority. In context-driven testing, we don't idolize "best practices."

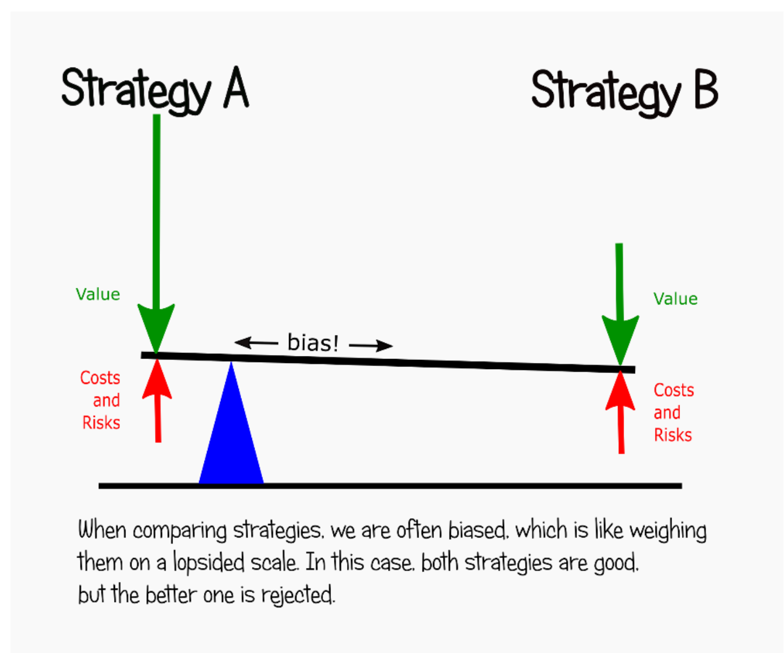

Figure 4

The answer to the question of how context drives tooling is: we read the situation around us; we discover the factors that matter; we generate options; we weigh our options, and we build a defensible case for choosing a

particular option over all others. Then we put that option into practice and take responsibility for what happens next. All along we are learning and getting better at this.

Building a test strategy and determining how to use tools to fulfill that strategy is an evolutionary process. No one who says "do it right the first time" has ever really done anything difficult right the first time. We become able to do things well partly via the experience of doing them badly. We often learn how to develop powerful, polished tools by developing cheap, disposable tools—and then throwing them away and applying what we've learned. Developing software also means developing our approaches to testing it.

Context-driven behavior tends to be highly exploratory because the practitioner is responsible, at every turn, for the quality of the work—and not just the immediate work, but also the overall strategy. If you are not just following instructions handed down from a boss or bureaucrat, then you have to evaluate the situation and frequently adjust your practices to get the best result you can. This also means that Context-Driven practitioners think not only about efficiencies, but about contingencies as well.

Approaching test tooling in a context-driven way means we don't play down the problems that tools have. We try to face them forthrightly. This can make us look rather pessimistic about certain ways of using tools, though, so it is a special challenge to remind ourselves of the benefits of tooling and to move toward the kind of tools that provide more of those benefits.

Do We Reinvent Every Wheel?

In practice, very little we do is designed from scratch. We collect and apply reusable heuristics by which we quickly solve common problems. We don't call these "best practices," because they aren't. They are patterns we find useful in particular situations, and we apply them mindfully. This involves looking to the cost, value, and contingencies of any given heuristic; and it involves ongoing re-evaluation of our process.

Invest in tools that give you more freedom in more situations.

So, how does a Context-Driven tester approach tools and their use? Well, there are no "best tools" in the Context-Driven world. Indeed, there are no dedicated "test tools" in the Context-Driven world. Any tool can be a test tool. Any tool might be useful. But we can suggest, all other things being equal, factors that make some tools more generally preferable. In good Context-Driven fashion, we acknowledge at least one exception for each heuristic:

1. **Tools that support many purposes are preferable to those optimized for one purpose.** Some tools are designed with specific process assumptions in mind. If you work in a context where those assumptions apply, you are fine. But what if you decide to change the process? Will your tools let you change? In changing contexts, tools that are simple, modular, or

adaptable tend to be better investments. Tools that operate through widely used interfaces and support widely used file formats are more easily adapted to new uses. Note that a tool may have only one major function, such as searching for patterns in text, and yet be a good fit for many purposes and many processes. *Exception:* If a tool happens to fulfill its one purpose far better than alternative tools, it might be worth the trouble of making room for it in your toolbox.

2. **Tools that are inexpensive (or free) are preferable to expensive tools even in many cases where the expensive tools are more powerful.** This is partly because of "sunk cost bias." The more money management pays to acquire a tool, the less acceptable it is to stop using the tool even if the tool is obviously unsuited for the purpose at hand. Furthermore, free tools invite us to experiment with different techniques. Experimenting is absolutely necessary in order to develop the skills and knowledge we need to make informed decisions about our processes. *Exceptions:* Remember there is more to cost than the purchase price. An apparently inexpensive tool may cost more in the long run if it requires extraordinary maintenance. Also, an expensive tool might be the only tool that has the special capabilities that you seek.
3. **Tools that require more human engagement and control are preferable to those that require less.** This is due to a syndrome called "automation complacency," which is the tendency of human operators to lose their skills over time when using a tool that renders skill unnecessary *under normal circumstances*. In order to retain our wits, we humans must exercise them. Tools should be designed with that in mind, or else when the tool fails, the human operator will not be prepared to react[6]. *Exception:* We may genuinely value the power and convenience that the tool gives us more than we value the skills and awareness that we lose in the process.
4. **Tools that are supported by a large and active community are preferable to those that are not.** The more people who use a tool, the more free support will be available and the more libraries, plug-ins, or other extensions of that tool. This increases the value of the investment in learning that tool, while reducing the learning curve. (The R language is a good example. It's a powerful and general purpose data analysis tool. Lots of researchers use R, lots of books about it are on Amazon.com, and there are hundreds of libraries that provide special capabilities beyond the defaults capabilities of the tool.) *Exception:* Just as in the case of expensive tools, sometimes the value you get from a tool is so important that it overrides concerns about support.
5. **Tools that can be useful to non-specialists are preferable to those that are not.** We're talking about tools that lower the cost of getting started; that afford ease of use; that don't depend on proprietary languages; that have lower transfer and training cost. Microsoft Excel and spreadsheets in general provide a good example. It is possible to use Excel in a very

---

[6] See Nicholas Carr, *The Glass Cage,* and Lisanne Bainbridge, "Ironies of Automation", *Automatica*, Vol. 19, No. 6. pp. 775 779, 1983.

specialized and sophisticated way, but there is a lot Excel can do for you, even if you have only basic skills with it. *Exception:* Sometimes it can be good for a tool to dissuade non-specialists from using it, because non-specialists may not be capable of using the tool wisely.

6. **Tools over which we have control are preferable to those controlled by others.** Good tools are at the very least configurable to your specific needs. An open source tool allows you to control every aspect of it, if you need to do that. Apart from the expense, commercial proprietary tools prevent you from adding new features and fixing critical bugs. Proprietary tools may be modified in ways that disrupt your work at inconvenient times, and you have no control over that schedule. *Exception:* Sometimes not having control over a tool is a good thing, because you are forced to use standard versions and configurations which allow you to share work more easily with others who use that tool.
7. **Tools that are portable across many platforms are preferable to those restricted to a single platform.** One aspect of context is the operating system or hardware platform. Cross-platform tools obviously work in a wider context. *Exception:* A tool may provide value that is important enough to offset its lack of cross-platform compatibility; or it may offer interoperability with similar tools on those other platforms.
8. **Tools that are widely (or easily) deployed are preferable to tools that aren't.** A primary problem of tool use is getting the tool in the first place. Some tools require complicated installation and configuration. Some tools may require special permission or expenditure. This can require negotiating with the IT department, managers, or co-workers. *Exception:* Some tools may be worth this trouble.

Intrinsic Testability

Certain aspects of the product design enable tool-supported testing. These include:

- **Observability**
- **Controllability**
- **Algorithmic Simplicity**
- **Decomposability**
- **Compliance to Standards**

Invest in testability[7].

The success of any tooling strategy depends in large part on how well your technology affords interactions with tools. This is why it pays to build testability into your products. From a tool perspective this means two big things: that your product is controllable by tools via easily scripted interfaces, and that its states and outputs are observable by those tools. For this reason, browser based products tend to be more testable while apps on mobile devices are less testable. Products with standard controls are more testable than products with custom controls.

[7] See http://www.satisfice.com/tools/testability.pdf

Consider making testability review a part of each iteration, and otherwise instill this thought process early in your projects. Any serious attempt to make tooling work must go hand-in-hand with testable engineering.

Let’s see tool-supported testing in action!

We now present you with three examples of how tools help testing. The first example involves no checking. The second is checking done partly with tools and partly by the tester. The third is fully automated checking through the GUI (and gives you an idea of why we generally avoid doing that).

These examples are worked from the perspective of the independent tester, rather than the developer. We will demonstrate by describing some of our testing of FocusWriter, a word processor with a minimalist design, intended to create a distraction-free environment that helps authors write novels.

CASE #1: Tool use without checking.

> *James wanted to test FocusWriter. He opened his browser, navigated to the gottcode.org Web site, downloaded FocusWriter, extracted it from its .zip file, and played with it.*

Are there any tools in use here, so far? Most testers would say no. But seemingly, according to our definition, the computer is a tool of some kind. Various parts of the computer are tools, such as the mouse, the monitor, and keyboard; hardware. The web browser James used to download the product is a software tool. The website he accessed is a tool, too. The Internet itself is a tool. Yet, no one feels that these are “testing tools”, nor that this activity is anything like “test automation.”

Why? Because none of these things are tools with respect to anything unique about testing. Instead, they comprise the fabric of ordinary computing; ordinary use of the product. When we speak of tools in testing, we do not mean the natural processes of using the product, but rather contrivances applied for the purpose of accelerating or enabling testing over and above ordinary human interaction with the product.

James opened SnapTimer and set a 15-minute rolling timer. He opened Evernote and started a new note titled “FocusWriter Test Session.” Then he Googled FocusWriter and thumbed through the top hits. In this way, he discovered that FocusWriter is an open source app.

The tools here are SnapTimer, Evernote, and Google. These are not part of the FocusWriter user experience, and they are not employed in simulation of what a user would do in the ordinary business of using FocusWriter. Therefore, these are bona fide test tools in this case. They are applied specifically for testing purposes, even though they may not have been designed for testing per se.

More tool use quickly followed:

1. *James located the source code on Github. (Google)*
2. *He used Git to download that code. (Git)*
3. *He unzipped it. (7Zip)*
4. *He opened a Windows command prompt. (CMD)*
5. *From the top of the source directory he used* grep *to search for the string "error." (grep)*
6. *He noticed this line in the output:* ***m_error = tr("Unable to open archive.");*** *On the conjecture that "tr" means translate, and therefore may be the formal mechanism for displaying localized message strings, he used a regular expression search to extract every associated string from the source. (grep with regex)*
7. *He extracted the strings themselves using this Perl program:* ***while(<>) { foreach (/tr\("(.*?)"\)/g) { print "$_\n" } }*** *(Perl with regex)*
8. *He used Notepad++ TextFX plugin to sort the result and eliminate duplicates. (Notepad++ with TextFX)*
9. *He grouped all commands together, all keyboard shortcuts together, all messages to the user together. (Notepad++)*
10. *At some point during this process, the 15-minute timer chimed, which alerted James to the need to update his notes and check in on his test charter. (SnapTimer and Evernote)*

Strings extracted from source code (as of step 9):

```
Left
Line Spacing
List all documents
Loading settings
Loading sounds
Loading themes
Longest streak
Manage Sessions
Margin:
Memo:
Minimum progress for
streaks:
Minutes:
Misspelled:
...
The requested session
name is already in use.
Unable to load
typewriter sounds.
Unable to open archive.
Unable to overwrite
'%1'.
Unable to rename '%1'.
Unable to save '%1'.
Unexpectedly reached end
of file.
...
&About
&Add
&Bold
&Change
&Close
```

We see plenty of tool use here, but most people would not call this "test automation." So what? It is tool-supported testing. Testers should think about tools as helping them in *any* aspect of testing.

This case also shows the power of a tester who already knows how to use basic, free, technical tools and possesses a foundation of technical knowledge sufficient to read and write code. Not all testers need that—but we suggest all testers need *access* to someone who does have that skill, such as a toolsmith on the team.

The use of tools in this case led to interesting results:

- Discovery of functionality referred to inside the product ("&Daily Progress") but not yet implemented.
- Discovery of error messages relating to previously unknown functionality.
- The basis for systematic testing of error handling.

CASE #2: Tool-support via patterned data generation for better coverage and a powerful oracle.

James frequently uses tools to generate special test data. One of his favorite tactics is called a "simplified data oracle." For FocusWriter he used that to test the Scene List and Filter functions.

The Scene List is a feature that allows the author to navigate, select, and move scenes more easily. A scene is a block of text delimited by a specific text marker, such as "##."

To test the Scene List we need a document that has scenes in it. That's easy. We create some text and put some scene dividers into it, as in Figure 5. This text will display in the Scene List as in Figure 6. Let's say this looks good. Let's say it is exactly what we wanted to see.

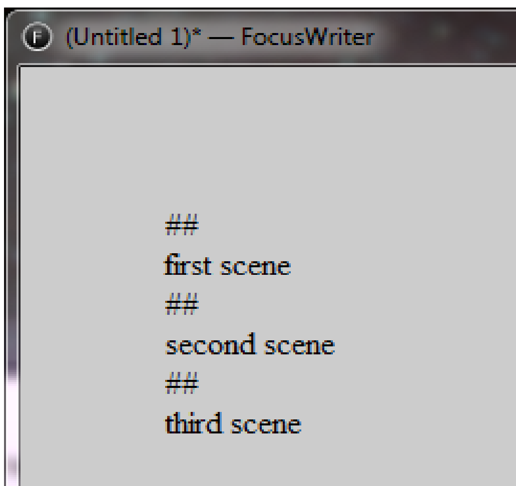

Figure 5

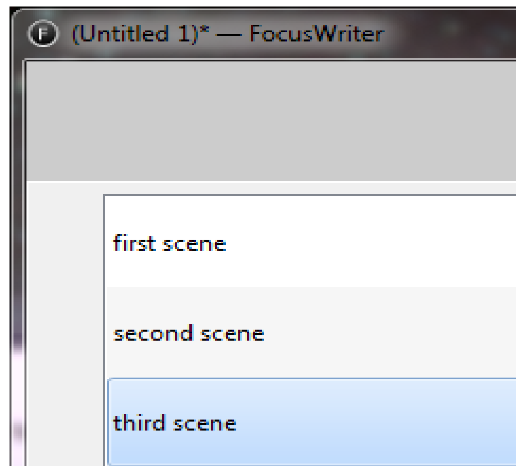

Figure 6

Now we could automate this as a typical output check. But tools can do so much more for us. So, James decided to write a program that would create thousands of scenes, and identify them in a specific way that would help us track whether they were in the correct order in the Scene List. In other words, it is a combination of a stress test and a correctness check (to be performed by a human tester) that helps us see if there is a bug in how the Scene List displays and sequences the scenes. It helps us test the scene divider handling for a variety of different divider strings (because the scene divider string is configurable) as well as scene filtering functionality.

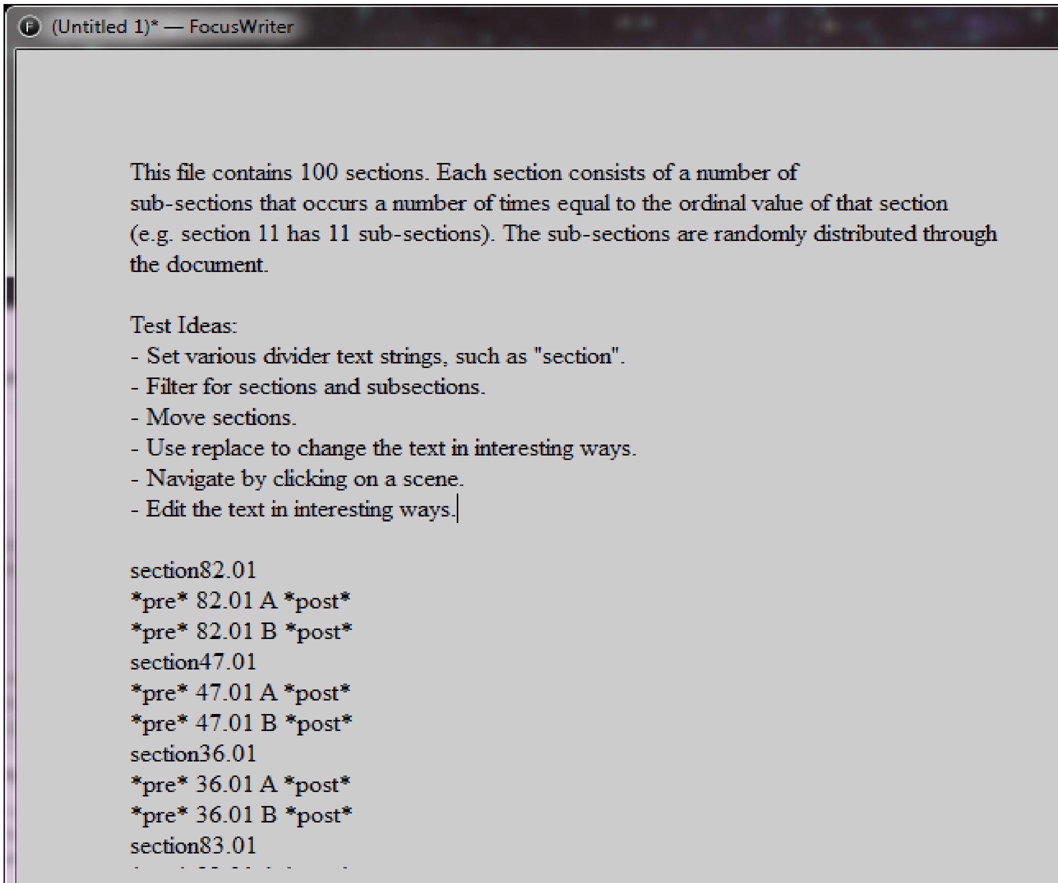

Figure 5

The program he wrote results in the file shown in Figure 7, which contains a total of 5,050 sub-sections. If the scene divider string is set to "scene" then the Scene List shows Figure 8.

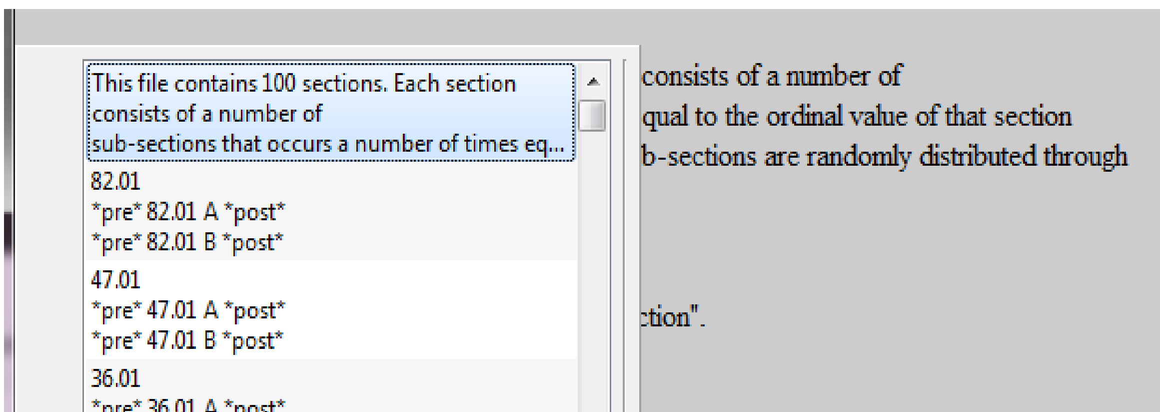

Figure 6

Now, by filtering on "07." it should pick out only seven scenes, wherever they are in the document, and display them in order in the Scene List panel. This is in fact what happens.

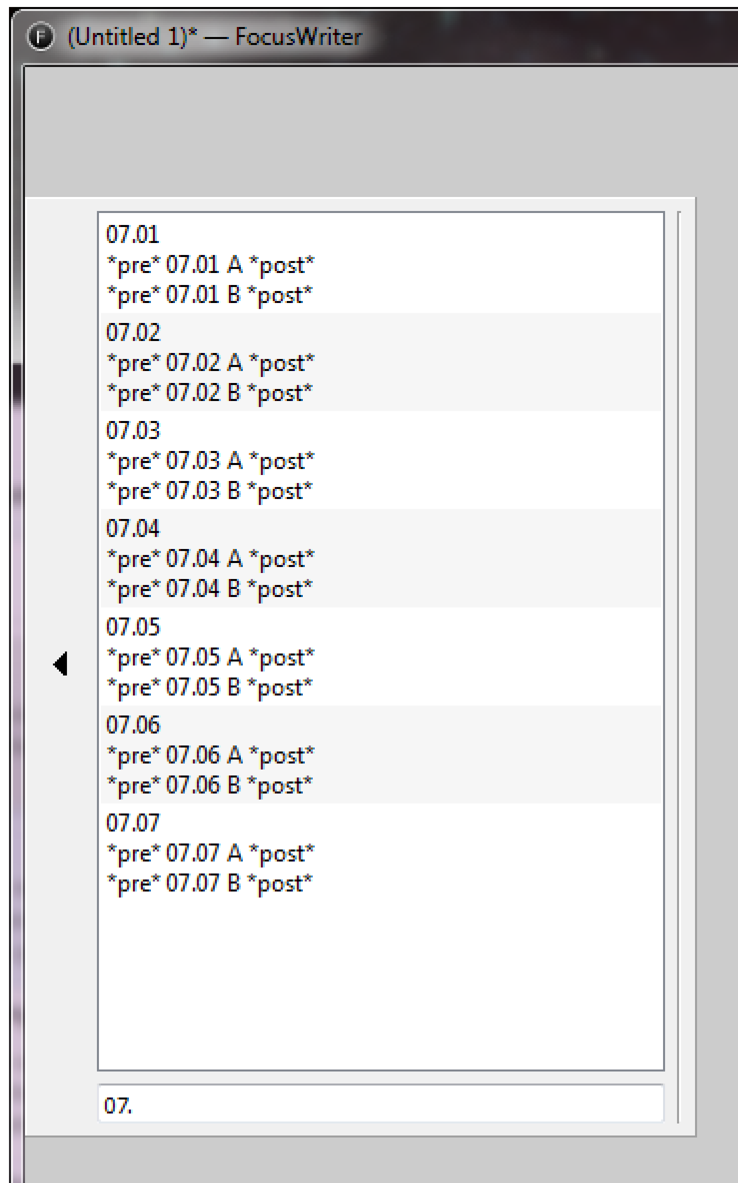

Figure 7

This is testing with a big boost from a tool. The tester can play. The tester can move scenes, filter, edit the document, or whatever. All the while, his test coverage will be deeper and his oracles sharper because of this patterned data. The tester is not limited to using the data, but can also edit the program that created the data to create even more interesting data. In fact, the version of the data you see, above, is the fifth refinement of the original concept.

CASE #3: Automated checking.

We wanted to demonstrate full-on automated checking, while at the same time staying with the FocusWriter example. FocusWriter does not provide an API for testing, which required us to automate through the GUI, and so the headaches began.

The high concept for the check was straightforward: perform a series of operations in FocusWriter that touch several features, change a document in a few ways, but result in the same output as we had at the beginning. A check that ends with the same state it began with is called *idempotent.* Idempotency is a useful heuristic because the process should be repeatable any number of times without regard for any progressive issues with system state—and if the state of the system interferes with the automation, that would be an interesting test result.

James proposed a process to be automated:

1. Delete any old temporary test files.
2. Start FocusWriter.
3. Load the Three Musketeers text file.
4. Search and replace all instances of lower case “e” with “~” (chosen since it does not appear in the text).
5. Save the file as type ODT.
6. Close the file.
7. Close FocusWriter.
8. Open FocusWriter.
9. Open .ODT file.
10. Search and replace all instances of “~” with lower case “e.”
11. Save as TXT in new file.
12. Compare original with new text document.
13. Log result.
14. Exit FocusWriter.

What about the unit level?

• • •

Automated, low-level checking is most famously embodied by the practice of “test-driven design” (which is really “output-driven design” but now it’s too late to rename it). We are not going to cover it here because it’s too big a subject and already gets so much coverage in the Agile world.

Automating low-level checks is a powerful practice that can improve testability and make quality easier to achieve. Like all checking it requires skill and forethought to pull off, and it is blind to many bugs that occur only in a fully integrated and deployed system. Still, it is generally much less trouble and expense than GUI-level checking.

This is designed to exercise saving (two kinds of files), loading (two kinds of files), starting, stopping, searching, and replacing. It comprises a bit of a stress test, because of the size of the Three Musketeers novel (1.3 million characters), but mainly it would be useful as a sanity check.

At James’ suggestion, Michael started to tackle the task using AutoHotKey, a Visual-Basic-like Windows scripting language. He soon ran into a problem: he was unable to query and confirm the state of the list box control by which the user chooses the file type. That obstacle and his unfamiliarity with AutoHotKey prompted him to switch to Ruby, with which he has a good deal

more experience. Ruby has several libraries that provide support for the Windows API. He soon found the RAutomation library which is billed as “a small and easy to use library for helping out to automate windows and their controls for automated testing”. Like many such open-source offerings, it is sparsely documented, but with a few minutes of experimentation, Michael was confident that he would be able to make sense of it and put it to work.

RAutomation proved to be intuitive and straightforward to use, but Michael quickly discovered that certain aspects of FocusWriter made automating the process tricky. Among other things, FocusWriter appeared to implement list boxes such that RAutomation (as AutoHotKey before it) could not determine the currently selected option for the current file type; Michael had to track this by other means. Saving a file would sometimes cause a confirmation dialog to appear, sometimes not. Several dialogs shared the same caption (“Question”), even though the prompts and options within were different. Frequently the script would initialize actions before the application was ready for them, requiring wait states of one kind or another. All of this required loops of experimentation, discovery, learning, and revision that, in the end, took hours. Among other things, Michael wished that he had been part of the development process for FocusWriter to appeal for better testability.

After much fiddling Michael succeeded in getting the process running reliably, but when James used the same script on his own system, it was not able to find and start FocusWriter! After an hour of investigating together, we abandoned Ruby.

We considered our problems so far. Perhaps what we needed was a tool optimized for interacting with the product via the GUI. We’ve heard of HP Unified Functional Testing and its predecessors from testers forever. HP says *“HP UFT software automates testing through an intuitive, visual user experience that ties manual, automated, and framework-based testing together in one IDE. This far-reaching solution significantly reduces the cost and complexity of the functional testing process while driving continuous quality.”*[8] To test this claim, we downloaded the trial version. After another full hour using the record and playback facility of HP UFT, we were able to get FocusWriter started, but could not get HP UFT to recognize the application window. It recognized Notepad, but there seemed to be something about FocusWriter (the fact that it is built with the QT toolkit?) that made it invisible to the HP tool. HP UFT would record a script, but then was not able to run its own script! We changed settings and edited the script in different ways, all to no avail.

Perhaps another five minutes or five hours would have gotten us past the problems with HP UFT. Neither of us are expert in the use of this *particular* GUI automation tool, and some experience with it might help us get around some of the obstacles. Perhaps our programming skills would accelerate our learning curve. Yet tools like this are often marketed in terms of “test automation without programming skills”. Here’s a typical example: “Test automation alleviates testers'

---

[8] http://bit.ly/1j9VUCL, retrieved October 30, 2015.

frustrations and allows the test execution without user interaction while guaranteeing repeatability and accuracy." Claims like these were being made 20 years ago. Meanwhile, the self-driving flying cars are still very much on the ground, with human drivers behind the wheel.

As a final effort, James used AutoHotkey to record a macro that performed the basic script. Success!

We found bugs…but not because we automated the check.

During this experiment in checking, we found that the final TXT file did not match the original one. That is all that checking can do: report some sort of inconsistency that must then be investigated.

Our subsequent investigation of the inconsistency led to two bugs:

1. FocusWriter writes ODT files in a manner that Microsoft Word complains is invalid (although it is apparently able to rescue the content).
2. FocusWriter reads ODT files incorrectly, causing one extra line and ten new spaces to be inserted wherever there is a line that begins with at least two leading spaces.

We reached our understanding of the first problem because we thought to use Microsoft Word and OpenOffice as "comparable product" oracles for the evaluation of the ODT file saved by FocusWriter. We reached our understanding of the second problem using a bevy of tools:

- **WinMerge** to analyze the text differences between the files
- **Frhedit** to analyze the hexadecimal differences between the files
- **Perl** to create and modify test files with various properties in order to test our hypotheses
- **7Zip** and **Notepad++** to examine the XML content of ODT files
- **Excel** to build a spreadsheet that predicted size changes
- **Wikipedia/Google** to study UTF-8 encoding

Note that automating the check had little to do with finding and investigating these bugs. It didn't save us any time. In this case, so far, the automated check is a cost without benefit. The check itself in its un-automated form—specified by a thinking tester, and carried out interactively during the process of automating it—simply gave us an indication of a problem. Then, skilled testers carried through with the investigation (with the help of tools) that systematically pared down potential factors to home in on the few that mattered.

The foray into automated checking took far more time than the first two cases. We learned things in the process of developing this automated check that might have made further checking easier to some degree, especially if our experience could inform better testability. Yet we wonder: if we were to create a library of checks, would the value of such checks match the development cost? The maintenance cost? The opportunity cost?

It may be that, sometime in the future, another bug will creep in to the product that this particular check will notice. If there are changes to this area of the product in the future, they will probably cause our check to fail irrespective of whether the failure is due to a bug in the product or in the script. If there are no changes in this area, the check will not fail, but probably also will not be worth running. It is often difficult, ahead of time, to choose which checks will be worth having automated, and which will turn out to have been white elephants. A lot of this is a matter of guessing about risk and change. This involves skill, experimentation and learning.

Why is automating interactions through a GUI so difficult?

After hearing this story, some "test automators" might claim that they don't have these kinds of problems, and if they did, they would be able to get around the problems easily. In saying so, they would be completely missing several points.

GUIs are designed for able-bodied humans, who don't have the same trouble finding and interacting with products as robots do. In fact, our tools fell victim to the same kinds of barriers presented to people who suffer from disabilities and yet try to use modern technology. Accessibility is a widespread problem in computing[9] for people and tools alike. Just as success with one accessible product does not disprove the existence of the accessibility problem in general, pointing to one trouble-free use of a tool to control an app doesn't disprove our general claim, which is this:

GUI-level scripting is fiddly. It's fussy. It fails suddenly in unexpected ways. It's notoriously problematic. It requires learning not only about the application and the tool, but also about how they will interact. We both learned this in the 80's and 90's[10], we've seen it ever since, and we are seeing it now.

What Every Toolsmith Knows

• • •

"GUI-level scripting is fiddly. It's fussy."

Beyond the accessibility issue, think about it. Even something as simple as saving a file, which is easily navigated by a human, becomes an exercise in pure pedantics when you attempt to program a machine to do the same thing. When you save a file, the application **may or may not** present dialogs that are distinguishable using a particular approach; it **may or may not** use libraries or controls that are recognized by the test framework you're using. On any given run, the application **may or may not** be pointed to the right directory; it **may or may not** ask you if you really want to overwrite another file; it **may or may not** refuse to proceed because that other file is locked by another process or because disk space ran out on your USB stick. The application **may or may not**

[9] http://arc.applause.com/2015/11/02/mobile-accessibility-end-user
[10] See James Bach, "Test Automation Snake Oil", http://www.satisfice.com/articles/test_automation_snake_oil.pdf

be interrupted by some other application during this process. It **may or may not** take an unusually long time to save.

Humans take all these eventualities in stride—we barely notice them unless they seem wrong! Not so for programs. Any possibility that has not been anticipated and not expressly programmed is a potential stumble for the script, which means another round of troubleshooting, debugging, and testing for the person trying to program it. Even if you think you could do better with FocusWriter using your favorite tool, we claim that the *kinds of problems* we have highlighted are not unusual in tooling generally, and that they are *normal* to GUI-level user simulation tooling, regardless of your skill level, industry, or product type.

Although Michael succeeded with Ruby and James succeeded with AutoHotKey, without more work, success took the form of rough prototype scripts. These are very brittle. The smallest change in the application, or in the saved state of the application, or in the data set may disrupt the script in a way that requires tuning or a complete rewrite.

You can make GUI checking more resilient in the face of product change, at a price…

As one of our reviewers, Ben Simo, noted, we can certainly make GUI checks less brittle in the face of change, but the very factors that make them less brittle usually make them less powerful, too, because we achieve greater resiliency by sacrificing certain sensitivities. For instance, we may filter out time stamps or user names, or we block out sections of screenshots. It may be okay to ignore the current user name when we are running the tests with accounts called "TestRobot6" or "TestRobot22" and want to use the same logic to check screens in both cases, but what if there is a legitimate bug whereby the user's name is wrongly displayed? Our modified check won't spot it. Adding such special case logic into the check code also increases the complexity of that code, which creates brittleness of a different kind: an increased likelihood that we will break the code when we try to improve it.

We may be completely successful in suppressing certain disruptions to GUI automation, but those same disruptions may give us information about the product that, as human users, we would easily understand as significant. For instance, we can implement a time delay in checking output from a process in order to assure that the process is complete before we attempt to process the output, but that means we won't notice if it gets progressively slower and faster in its response times *within* that time delay. Testers don't just numbly watch the world go by. Real testers are not just idle product tourists—we critically analyze what we see. But if we outsource our "seeing" to the computer, we cannot be critical of what *it* sees, and the computer doesn't know *how* to be critical.

As you navigate these troubles, you will probably be caught up in a more insidious pattern: GUI-level checking distracts testers from performing deep tests that examine complicated or subtle functional behaviors. This is because you have to keep the checking simple. If you pour complex, interesting data and interactions into your checks, you will create huge headaches for yourself in

coding and maintaining it. Imagine reproducing, in code form, just five minutes of your typical use of your computer. Yikes. This is why GUI check designers focus on superficial interactions and easily parsed outputs. It's economically viable. It will allow you to add more "test cases" to your "test suite", but what it accomplishes is likely to be shallow testing. At the same time, all this effort presents opportunity cost that robs you of time for deeper testing.

There are contexts in which automated checking is likely to be cheaper and more powerful. Products that are built with testability in mind—scriptable interfaces and log files that can be easily parsed—tend to be more amenable to automated checking. Products that have simpler forms of input and output are easier to check programmatically. Automated checks closer to the developer's current task can afford quick change detection, fast feedback, and simpler repair. Well-built, "unbuggy" products can be much easier to automate and to check.

These points we are making are not new. The first Los Altos Workshop on Software Testing, which was the first organized gathering of the not-yet-named Context-Driven School of software testing dealt with the problems of automating the interaction of an application through a GUI—and that was way back in late 90's[11].

Automating actions is a tactic. It should not be a ritual.

In the Context-Driven world, we reject ritual. We embrace problem-solving. But this attitude is only valid if problem-solving matters. Too often, automation (in both senses—artifacts and enterprise) is pursued as an unquestioned good; stuff that dazzles people even when it accomplishes little. Large tool companies and consulting firms aren't much help, either: it's not in their interest to help you see a simpler, cheaper, more flexible way of doing things.

If your testing doesn't really matter, except as a display for public relations purposes, then maybe rituals are acceptable— but that cannot be so if your intention is to find important bugs before it is too late. To fulfill *that* mission, you must develop an appreciation of the full spectrum of tools and their applications to your work. Context-Driven testers apply tools in powerful ways to get testing done!

[11] Kaner, Cem, *Improving the Maintainability of Automated Test Suites*, p. 10, http://www.kaner.com/pdfs/autosqa.pdf