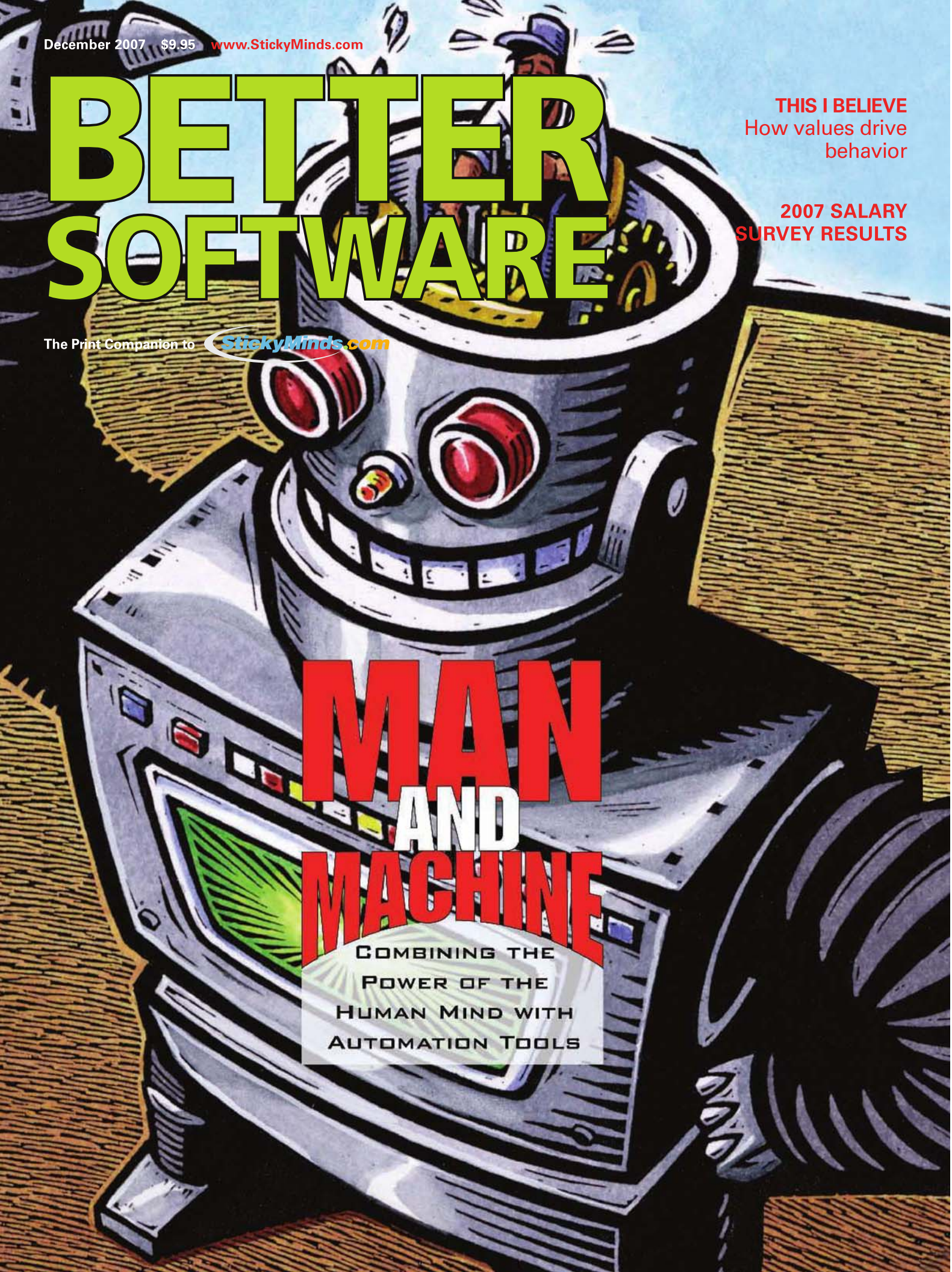
December 2007 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

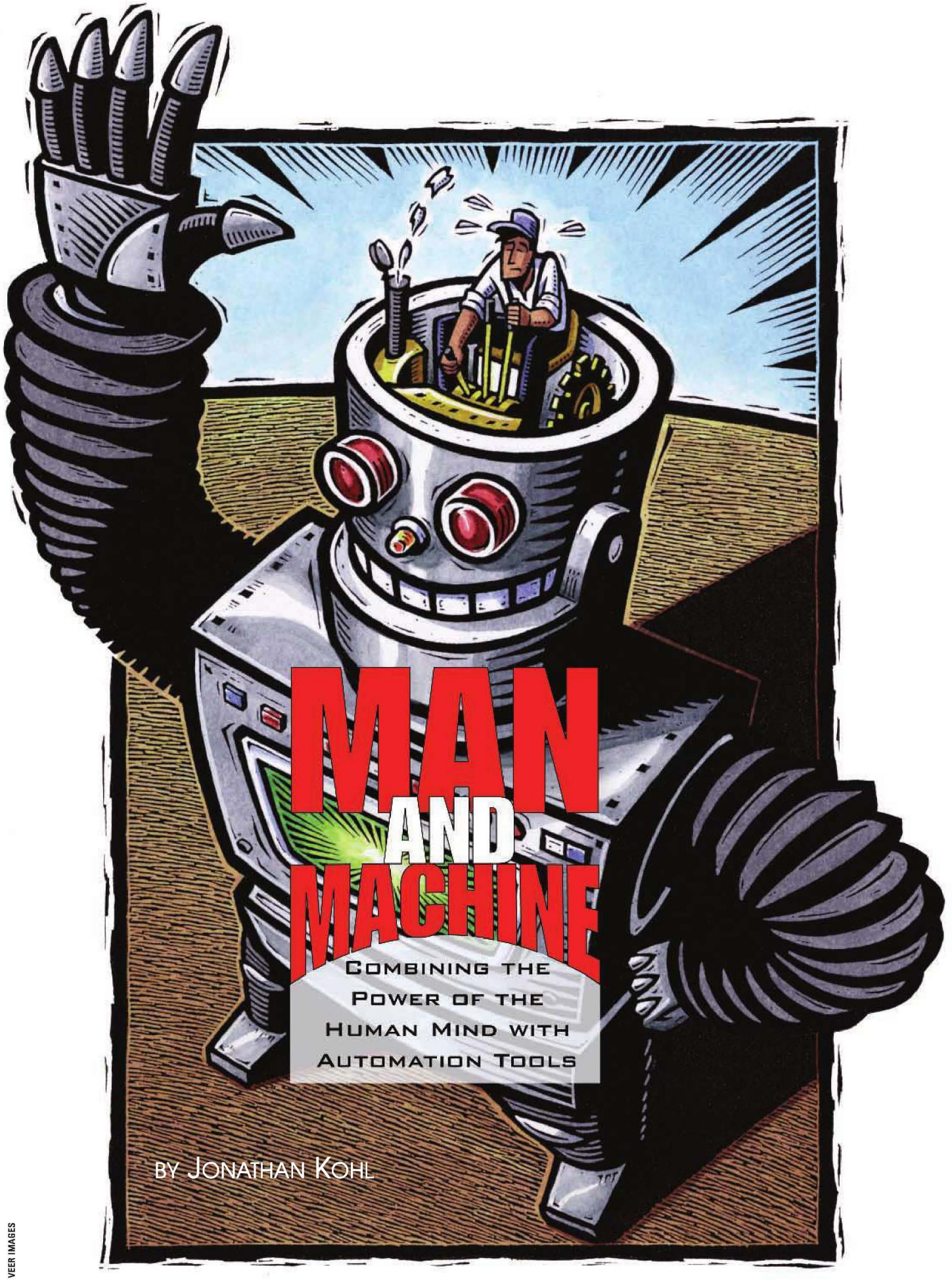
THIS I BELIEVE
How values drive behavior

2007 SALARY SURVEY RESULTS



MAN AND MACHINE

Combining the Power of the Human Mind with Automation Tools



MAN AND MACHINE

Combining the Power of the Human Mind with Automation Tools

BY JONATHAN KOHL

VEER IMAGES

I like to walk to work. I enjoy the scenery, the exercise, and the time to think about problems I'm working on. Some of my best ideas come from these walks—away from the computer, out in the fresh air. Many of these ideas are triggered by observations that lead me to creative solutions.

But walking can be slow, and I can be thrown off schedule if I stop to watch wildlife or to enjoy a sunrise over the mountains. So, when I'm in a hurry, I use an alternate form of transportation. A bus or train trip still involves some walking, but if I need to get to a meeting on time, I can get there much more quickly than if I walked the entire way. I rarely drive to work; I end up so engrossed in the task of driving that I have little time to think and observe my surroundings. However, if someone else is driving, I notice a lot more. For example, I recently took the train into work, and on the way in I saw a bluff in the river valley I had never before noticed. I've passed the river valley many times, but that day on the train I was able to observe the scenery from a completely different perspective—without the distractions I face when walking or driving.

Similarly, when testing, I look for tools to assist me on the task I need to accomplish. If I want to thoroughly investigate the software I'm testing, I do more manual testing and analysis. If I need to complete tasks quickly or repeat tasks that require little thought, I use some form of automation. I call this hybrid of manual and automated testing "interactive automated testing." Instead of viewing test automation as an effort to replace all manual tests, this style of automation focuses on extending the abilities of the tester with an automation tool. It can also provide a different perspective on the software I am testing.

THE ABILITY TO PULL MYSELF OUT OF AN IMMERSIVE TESTING ENVIRONMENT AND VIEW MY TESTING FROM A DIFFERENT PERSPECTIVE IS A POWERFUL THINKING TOOL.

Using Automation with Exploratory Testing

When I started out as a test-automation specialist, my goal was to automate as many testing tasks as possible. My team thought that running tests unattended was ideal, so we wrote scripts to automate complex, manual test suites. However, some tasks were difficult to automate reliably and were more trouble than they were worth. Many times our unattended scripts would miss bugs we hadn't thought to program for or would fail at tasks that are simple and routine for humans but difficult for a computer. Our test automation ended up with some degree of human intervention, at least some of the time. While this felt like a setback, our automated test harness was running quite reliably, and when we did need to step in and help out manually, we tended to discover important information about the software we were testing. We noticed that our unattended automated tests discovered far fewer bugs than did our combination of manual and automated testing.

The combination of manual and automated testing involves trade-offs. Unattended automated test suites can be run frequently, quickly, and when there are no human testers around. Unfortunately, machines aren't intelligent, so they can't observe and investigate suspicious behavior, vary their testing activities, or report results the way a human can.

Sometimes when performing exploratory testing, I want to focus on an area deep within an application. If setting up the exploratory test requires a significant number of steps, I will use a tool to automate navigation through the graphical user interface (GUI). Since I am creating an automated test setup that I can use for different tests, I might call this tool a "text fixture" for my exploratory testing. Once I have the navigation automated, I run my completed test fixture, watch it play back on my monitor, and then take over manually when I get to the section where I want to focus my testing.

Automating exploratory test fixtures or the setup required to run a test can be an effective tool to enhance manual testing. You get the speed, precision, and repeatability of an automated test, coupled with the power of a curious human tester observing and investigating the application. Not only does this help testing become more efficient, but being able to watch the application work without any distractions from your program interaction also can provide a different perspective. When I'm not physically operating the application by typing or using a mouse, I see things differently. I can focus on a particular area of the screen more easily or watch for patterns that I miss when I'm busy typing or clicking.

The ability to pull myself out of an immersive testing environment and view my testing from a different perspective is a powerful thinking tool. Watching an automated test run is one way of pulling yourself out of your manual-testing environment. I've often seen previously unnoticed behavior in an application when running a test fixture, stopped it, taken over manually, and found an important bug. I find this perspective to be a good source of testing ideas.

Observation, Investigation, and Automation

Automated test fixtures were a great kick-start for my exploratory-testing sessions, but I soon learned to incorporate other forms of automation in conjunction with manual testing. In one case, a high-profile, intermittent bug was passed on to the test team. We were under time pressure to track it down and help the programmers fix it. I used an automated test fixture to navigate for me, and I performed several manual tests in the area of concern. Realizing it would take a significant amount of time to explore my current test ideas, I modified my test fixtures to create my test data and to repeat a test in a loop. I suspected that the bug appeared to be intermittent because two users were simultaneously performing actions in one area of the application. To simulate this, I ran one automated script to create test data, another to repeat steps in a loop, and then I manually tested on another machine. I managed to re-create the bug because it required two

simultaneous users accessing a common administration feature with just the right kind of test data.

Recently, I was manually testing a Web application feature that allowed long text-field inputs. Rather than count and type in characters by hand to test the limits of the error-handling code, I used a tool to generate test data. PerlClip is an excellent tool for this task. One input type it helps you generate is "counter strings," which are self-counting strings of a user-determined length. If I tell PerlClip to create a test string that is 255 characters in length, it quickly generates it and adds it to the Windows clipboard. To use it, all I need to do is paste it into the application.

PerlClip helped speed things up for me, but my tests also involved logging out, shutting down the browser, and starting over after each text input submission. This was getting a bit redundant and repetitive; I was getting distracted, logging in and navigating through the application to get to the section I was testing. Sometimes these distractions are useful because I can explore interesting behavior outside the current testing focus, but in this case, I needed to minimize distraction—I needed speed and precise navigation. Since I was in a Java environment, I used the test automation tool Web Application Testing in Java (WatiJ) to help out. WatiJ drives Internet Explorer in much the same way an end-user would, and it plays back scripts on your monitor.

In minutes, I created a simple WatiJ test that logged in with my test user account and navigated through the application to the section I wanted to test, and then it stopped. At that point, I took over the Web browser and entered my PerlClip-generated test data. I executed my tests much more quickly and had the added benefit of observing the application without needing to type or use the mouse.

This new observation perspective paid off. When my WatiJ test started a new Web browser instance and attempted to log in, I noticed the processing time for login attempts varied slightly on each script run. This looked suspicious—a potential source for a bug. When I was the one typing, I hadn't noticed this because it was only a slight difference. I took note of this behavior to remind myself to explore it further when I had completed my current task.

Once I had completed my testing task, I decided to investigate the behavior of the login page. Since the login attempt page response times varied slightly, I knew I needed to repeat a similar test several times. I also wanted to design this test from an exploratory testing perspective, so it needed to be flexible enough for me to change on the fly and to vary inputs. I created a new WatiJ test case that would be run by the JUnit unit-testing framework (see the StickyNotes for a link). I could now control the automated test and observe the application from a different vantage point.

```
public void testInputLength() throws Exception {
    ie.textField(name, "user").set(System.getProperty("inputLength.10"));
    ie.textField(name, "password").set(System.getProperty("inputLength.10"));
    ie.button("Login").click();
    assertTrue(ie.containsText("Login Failure"));
}
```

Listing 1

Once I had a WatiJ test that would enter in a username and password, click the login button, and verify that a login error message occurred, I began adding test data. I stored various kinds of PerlClip-generated test data in a Java ".properties" file, starting with counter strings of various lengths. I had some short strings and some very long strings—much longer than a typical username and password. I added a setup method to my JUnit test class that would create a new Internet Explorer browser instance, navigate to the login page, and load the test data. Listing 1 shows the test case.

This JUnit test uses the WatiJ library to run Internet Explorer, enter test data in the username and password fields, and click the login button on the login page. In this case, it uses test data, which is called "inputLength.10," meaning the size of the test input is ten characters in length. It then does a JUnit assertion—a check on the result of the test. In this case, it asserts that an exact match of the error message "Login Failure" occurs. The assertion will catch any slight deviation in the message that I might miss while rapidly repeating the test.

The test ran and passed as expected, so I decided to change the test to use a longer input string in each field. I had an input string that was 255 characters long, so I changed the test, saved it, and ran it again. I noticed that the time to process this login attempt was much longer than the previous attempt. To explore further, I created a group of similar tests in my JUnit class. I also added a "teardown" method that closed Internet Explorer after each test.

The timing differences were obvious when I used very small strings and very large strings. I added a new test case that had a counter string of 5,000 characters, which was a deliberate input penetration attack. This time, the application failed, with a Java exception stack trace. Modifying my set of tests, I was able to ascertain within a couple of minutes that an input attack of 4,000 characters or more would cause an overflow error. I wasn't sure why the application was slowing down, but noticing this fact and designing tests using my automation tools helped me quickly find an important bug.

A programmer came by while I was exploring my test results and saw the error message popping up on my screen. He pulled up a chair and sat with me. "Cool. Are you using JUnit?" he asked as I demonstrated the failure for him. He asked for the test case so he could run it on his machine. Once he had fixed the code so that the JUnit "red bar" (test failure) had turned to a "green bar" (test passed), he called me over and demonstrated it. We improved the test, checked it into source control, and started a new build.

Using the computer as an automation tool, combined with my observation and investigation skills, we quickly got to the bottom of a previously unnoticed bug.

Here are some automation ideas to help complement manual testing:

- Create test fixtures to automate test setup using a GUI automation tool
  - Program navigation
  - Test data population
  - Workflow creation
- Automate the creation of test data and populate the application with it
  - I like using interfaces behind the GUI, such as Web services or database interfaces, as they tend to be less susceptible to change and often are much faster at processing
- Simulate:
  - Automate repetitive use of commonly used features or workflows
  - Create light user load for usability testing of client-server applications
    - This can be done on multiple machines using a GUI or on one machine using multiple threads and the protocol the application uses
- Automate build deployments, application installation, and file management
- Monitor:
  - Visually with diagnostic tools
    - e.g., the Unix "tail" command is useful to watch server logs in real-time
  - Create a log file or other activity watcher that alerts you if certain conditions occur when testing
    - Brian Marick's *Everyday Scripting with Ruby* has an example "watch dog" script. See the StickyNotes for more information.

The combination of a good automation test tool and the investigative mind of a tester was the recipe needed to find this bug. Furthermore, since I was using the same language and tools that the programmers were using, it was much easier for us to collaborate.

Blurring Manual and Automated Testing

Sometimes, test automation advocates scoff at manual testing, while manual testers don't see how running unattended automated tests is an accurate reflection of the work they do. Often, this divide in thinking prevents utilizing the skills of both groups. However, automated testing and manual testing need not be contradictory concepts. Interactive automated testing bridges this gap and provides an opportunity to test applications in different ways.

Different testing tools can help provoke different kinds of test ideas. Marshall McLuhan said: "We shape our tools and thereafter our tools shape us." (see the StickyNotes for a reference). Before I used tools to enhance my manual testing, I didn't adapt to changing conditions or observations as easily or as quickly. Sometimes I didn't even see a change or an error condition that was being thrown on the backend because I was only testing through the GUI and wasn't using a tool to monitor more closely. When I did see a problem that required a lot of setup and repetition to investigate, it required more time and thought to create and execute tests. Now, I tend to use tools to monitor servers, query data that is being used by the application, and help execute different kinds of tests. My testing has been likened to a cyborg—part human, part machine.

Here is a recent example. A tester was running SQL scripts on the database backend of a Web application. She noticed that some of the data was duplicated—seemingly intermittently—and she asked for my help. This was a good step; she was using an automated tool for monitoring. She thought that the bug might occur when running similar types of tests. The only problem was the tests required an involved workflow with a lot of fields for an end-user to fill in on

several screens, so running a test manually took a lot of time. It was also susceptible to a lot of variation in inputs, so the tester had trouble remembering exactly what she had done. Since I had Ruby and Web Application Testing in Ruby (Watir) installed on my machine, I immediately began thinking about tests I could design using that tool to help. I began designing a test, much like an experiment, where I would repeat a test case ten times to see how many times the intermittent bug occurred. I wanted to minimize variability and have control over my inputs, and I wanted to repeat a similar test several times quickly. I created a Watir script simulating the user workflow, with all the user inputs stored as variables so I could control them more easily. This took about fifteen minutes to create. I then ran the test several times under my supervision, watching the script play back on the screen. We then monitored how often the bug occurred and found the data duplication that was occurring with this workflow. In short order, we had the information we needed for the programmers to begin investigating a fix. We also had a script we could reuse during the bug fix process and for other testing tasks.

The tester was surprised to see me open up a programming environment at first, but when I explained what I was doing, she was intrigued. "Can I have the script? I want to learn how to do that for similar cases!" In short order, I helped her install Ruby and Watir and got her off and running, creating her own scripts. I was suddenly struck by how much Watir shapes my Web-application testing. When the Watir project began about three years ago, I was a contributor on the project and had influence on its design, primarily through features I needed for test automation. Now, it and other tools are strongly integrated into my Web-testing work. I use some tools for user simulation and others for application monitoring, and I rely on the feedback of both to guide my testing. Now that I rely on the tools, they have become enmeshed with my manual testing activities. It can be difficult to divide my testing into either only manual testing and only automated testing. I have found that I can observe and investigate potential problems in real time during testing much more easily this way.

Combining the power of the human mind with automation tools helps fuel observation and discovery. When you combine these testing activities, you'll be surprised where your testing journey will take you. **{end}**

Jonathan Kohl is a software testing consultant with Kohl Concepts Inc., based in Calgary, Alberta, Canada. Jonathan writes about and speaks on software testing. Read more of his work at www.kohl.ca. Contact Jonathan at jonathan@kohl.ca.

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- JUnit testing framework
- References
- *Everyday Scripting with Ruby*
- Influences