Article development led by **acmqueue**
queue.acm.org

Automated usability tests can be valuable companions to in-person tests.

BY JULIAN HARTY

Finding Usability Bugs with Automated Tests

IDEALLY, ALL SOFTWARE should be easy to use and accessible for a wide range of people. However, even software that appears to be modern and intuitive often falls short of the most basic usability and accessibility goals. Why does this happen? One reason is that sometimes our designs *look* appealing so we skip the step of testing their usability and accessibility—all in the interest of speed, reducing costs, and competitive advantage.

Even many large-scale applications from Internet companies present fundamental hurdles for some groups of users, and smaller sites are no better. We therefore need ways to help us discover these

usability and accessibility problems efficiently and effectively.

Usability and accessibility are two ways of measuring software quality. This article covers several ways in which automated tests can help identify problems and limitations in Web-based applications, where fixing them makes the software more usable and/or accessible. The work complements, rather than replaces, other human usability testing. No matter how valuable in-person testing is, effective automation is able to increase the value of overall testing by extending its reach and range. Automated tests that are run with minimal human intervention across a vast set of Web pages would be impractical to conduct in person. Conversely, people are capable of spotting many issues that are difficult to program a computer to detect.

Many organizations do not do any usability or accessibility testing at all; often it's seen as too expensive, too specialized, or something to address after testing all the "functionality" (which is seldom completed because of time and other resource constraints). For these organizations, good test automation can help in several ways. Automated tests can guide and inform the software development process by providing information about the software as it is being written. This testing helps the creators of the software fix problems quickly (because they have fast, visible feedback) and to experiment with greater confidence. It can also help identify potential issues in the various internal releases by assessing each release quickly and consistently.

Some usability experts find the idea of incorporating automated tests into their work alien, uncomfortable, or even unnecessary. Some may already be using static analysis tools such as Hera and Bobby to check for compliance with WCAG (Web Content Accessibility Guidelines; <http://www.w3.org/TR/WCAG20/>) and Section 508 (<http://www.access-board.gov/sec508/guide/1194.22.htm>), but

not yet using dynamic test automation tools. As a result, they catch some problems but miss others (which was the case for several of the examples given later in this article).

One aim of this article is to encourage readers simply to *try* applying some automated tests to see if they help uncover issues that may be worth fixing.

Why is Usability and Accessibility Testing Difficult?

It's clear that companies today are not

doing enough usability and accessibility testing, and part of the reason is that it can be difficult to accomplish. Here are a few reasons why.

It's often difficult to understand users' frustrations when interacting with software, especially when their needs differ from ours. For example, if I'm a 20-something with good eyesight and mobility, an immensely detailed multilayered user interface might suit me well. But what about users with different abilities? Will they find it frustrating or possibly even

unusable? Finding ways to accommodate a wide range of users is challenging, particularly for corporations that have problems accepting offers of free help from groups such as blind users and people with motor impairments. Although rejecting such help may seem illogical, the logistics of preparation, transport, adapting the office environment to accommodate the visitors, and so forth, may discourage those who already have lots of demands on their limited time and resources.

There are a range of devices that bridge the gap between user and application—from mice and keyboards, to screen readers, to specialized equipment that adapts the user interface for people with severe impairments. Unless we have had experience with these tools, it is difficult to conceive of how they work in practice and how they affect the user's experience with *our* application. Furthermore, developers of UI tools seldom provide functional interfaces to support test automation or screen readers—making both challenging to implement.

In my view, usability testing is not inherently difficult, but it tends to be time consuming and hard to scale when it requires human observation of the people using the software being assessed. Also, because software developers seem to find the work fiddly or extraneous, they may fail at the first hurdle: deciding whether the testing effort is worth the investment.

Difficulties in Test Automation

In addition to the basic challenges in usability and accessibility testing, there are also challenges in developing good automated testing frameworks. Although pockets of interesting academic research exist that focus on automated testing, as an industry we've found it difficult to translate

academic work into practical value. Sharing of knowledge and tools from academia has been limited, as companies need first to pay for access to many of the papers and then translate the formal structure of those papers into something that makes sense to them. These solutions must also address the expectations implicit in the question, "Will this solve some of my immediate issues today?" Complicating things is that commercial automated-testing tool providers tend to guard their tests and methods, as they perceive this to be to their competitive advantage.

Many test-automation frameworks are not used regularly, to the extent that practitioners actually run the automated tests again and again. Furthermore, in my experience many aren't even used by their author(s). They are written once—perhaps so the authors get good performance reviews from their managers, rather than providing practical value to their projects—and then fall into disuse, as no one sees the value in the work or wants to support it. Longer-term test automation tends to suffer from fragility, poor maintainability, and inadequate software engineering practices. Good test automation is like good software development, requiring similar skills, practices, and pas-

sion to create and maintain it.

Another difficulty in test automation is finding bugs that "bug" people to the extent they are deemed worth fixing versus bugs that will be discounted because users are unlikely to encounter them or because developers don't see the value of fixing them.

Still another challenge is integrating all the various Web browsers with the test-automation software. Each browser is distinct and needs special code so it can be used with automated tests. This code can be fairly complex and challenging to write. Developers have started various projects using a single browser only to discover that the overhead of trying to extend their work to additional browsers is significantly more complex and time consuming than they are prepared for.

Finally, many test-automation tools still require their users to have technical and programming skills (for example, Java, Maven, JUnit, IDEs, among others) to write the tests. For open source projects the initial learning curve may be too steep to get the software to run on your computer. Some companies try to dumb down the test automation so people without a programming background can write tests, but these attempts often cause more harm than good.

Examples of Automated Testing

In 2009, I helped test several global software applications at Google. They were constructed using GWT (Google Web Toolkit), a very powerful development framework that allows developers to create cross-browser Web applications and hides many of the complexities from the developers. The resulting applications looked sleek and modern, and each had a user base of millions of people.

Development teams estimated GWT saved person-years of work in getting their applications into production. The way in which the teams were using GWT, however, resulted in several side effects that ended up generating usability and accessibility problems, such as broken keyboard navigation and poor support for screen readers. We discovered similar issues for other applications that used other frameworks and approaches, indicating these problems

Figure 1. Screenshot of Google Search keyboard navigation test.

might be widespread and prevalent throughout the industry.

My main goal was to determine whether we could create automated tests that would help identify potential problems that may affect *quality-in-use* for groups of users in terms of dynamic use of the software. As mentioned earlier in this article, several standards (for example, Section 508) and guidelines (for example, WCAG) aim to help address basic problems with accessibility, and a plethora of software tools are available to test for Section 508 and WCAG compliance. None, however, seemed to focus on quality-in-use of the applications.

Furthermore, my work needed to provide positive ROI (return on investment), as well as be practical and useful.

Testing Keyboard Navigation

One facet of usability and accessibility testing is keyboard input and navigation (as opposed to relying on a mouse or a touch screen). I decided to focus on finding ways to test keyboard navigation using automated software tools. The work started with a simple but effective heuristic: when we tab through a user interface, we should eventually return to where we started—typically, either the address bar in the Web browser or the input field that had the initial focus (for example, the search box for Google's Web search).

The initial test consisted of about 50 lines of Java code. It provided a highly visible indicator of the navigation by setting the background of each visited element to orange; each element was also assigned an ascending number representing the number of tabs required to reach that point. The screenshot in Figure 1 shows an example of navigating through the Google Search results. The tab order first works through the main search results; next, it tabs through the ads on the right, and then the column on the left; the final element is the Advanced Search link, which is arrived at after approximately 130 tabs! The code tracks the number of tabs, and if they exceed a specified value, the test fails; this prevents the test from running indefinitely.

This test helped highlight several

One aim of this article is to encourage readers simply to try applying some automated tests to see if they help uncover issues that may be worth fixing.

key issues such as *black holes*, Web elements that “swallow” all keystrokes. It also helped identify Web elements that were unreachable by tabbing through the page. Our success was measured by the percentage of bugs fixed and the reduction in keystrokes needed to navigate a user interface.

Several GWT frameworks include custom elements such as buttons. Developers can attach a JavaScript handler to these elements to handle specific keystrokes. We discovered a serious bug whereby JavaScript mistakenly discarded all the other keystrokes (apart from the specific ones it was designed to handle). This meant that once a user navigated to that button, he or she was unable to leave using the keyboard (tab characters were being silently discarded).

We used a heuristic in the automated test that assumed that if a user pressed the Tab key enough times, the user should eventually return to where he or she started in the user interface. The test included a “maximum number of tabs” parameter. We set this to three times the number of Web elements on the Web page as a balance between making sure we didn't “run out” of tabs before reaching the end of a legitimate page and the test continuing forever if we didn't cap the number of keystrokes. If the test failed to return to the initial element, it failed. This test was able to detect the black hole caused by the erroneous JavaScript. The problem was finally fixed by changing the code in the underlying custom GWT framework.

The second problem we discovered was a “new message” button that was unreachable using the keyboard. This was embarrassing for the development team, as they prided themselves on developing a “power-user” interface for their novel application. One aspect of the test was that it set the background color of each Web element it visited to orange. We were able to spot the problem by watching the tests running interactively and seeing that the “new message” button was never highlighted. We were able to spot similar problems by looking at screenshots saved by the test automation code (which saved both an image of the page and the DOM (document object model) so we could visualize

the underlying HTML content).

The third problem was more insidious and initially more difficult to detect. GWT used a hidden IFRAME in the Web page to store the history of Web pages visited by the user (so the user could navigate with the browser navigation controls such as “Back”). We discovered, however, that one of the initial tab characters was directed to the hidden IFRAME. This was confusing for users, because the cursor disappeared, and it was also mildly annoying, as they had to press an additional Tab character to get to where they wanted in the user interface. Once the problem was uncovered, the fix was easy: add a TABINDEX=“-1” attribute to the hidden IFRAME.

The next heuristic we considered was that the sum of the number of tabs should be identical for both forward (Tab) and reverse (Shift+Tab) keystrokes. The first part of the test used the same code as that used for the initial heuristic, where the count of tabs issued is incremented for each element visited. Once the test reached

the initially selected element, it started generating the Shift+Tab keyboard combination, which caused the navigation to go in reverse. Again, the number of Shift+Tab keystrokes was counted. Each time an element was visited, the value set in the title property of that element was added to the current value of the counter. The sum of the tab-orders should be identical for every element visited. If not, there is a hysteresis loop in the user interface indicating a potential issue worth investigating. Figures 2 and 3 show the tab counts for each element visited. We can see that each pair of values for a given Web element add up to nine (for example, Button B’s counts are: $4 + 5 = 9$; and Button A’s counts are $6 + 3 = 9$). So this test passes. [Note: the figures do not include extra tabs required for the browser’s address bar, etc.]

The final heuristic was that the flow of navigation should match a regular pattern such as down a logical column of input fields and then right and up to the top of the next column.

Figure 4 shows two typical flows.

Here we can detect whether the expected pattern is being followed by obtaining the (x,y) location of each element on the Web page. The pattern may be explicit (if we know what we want or expect) or implicit (for example, based on how similar Web pages behave). A tolerance may be used to allow slight variations in alignment (where we consider these to be acceptable).

Our automated tests rely on WebDriver, now part of the open source Selenium test-automation project and known as Selenium 2.0. WebDriver aims to interact with a given Web browser as a person would; for example, keystrokes and mouse clicks are generated at the operating-system level rather than being synthesized in the Web browser. We describe this as generating native events. Sometimes WebDriver cannot generate native events because of the technical limitations of a particular operating system or Web browser, and it must compensate by using alternative in-

Figure 2. Testing for forward tab counts.

Figure 3. Testing for reverse tab counts.

Figure 4. Typical navigation flows.

put methods. For the keyboard navigation tests, though, generating native events is essential to establishing the fidelity of the tests.

WebDriver works with the majority of popular desktop Web browsers, such as Firefox, Internet Explorer, Opera, and even includes the Web browsers on Android, iPhone, and BlackBerry devices. This broad reach means we can run our tests on the most popular browsers, which helps increase the usefulness of the tests.

Finding Layout Issues

Layout problems are an area that can adversely affect a user's perception of an application and may indirectly reduce its usability by distracting or frustrating users. There are numerous classes of problems that can cause poor layout, including quirks in a particular Web browser, mistakes made by the developers and designers, and poor tools and libraries. Localizing an application from English to languages such as German, where the text is typically more voluminous, is a reliable trigger for some layout issues. Many of these problems have been challenging to detect automatically, and traditionally we have relied on humans to spot and report them.

This changed in 2009 when I met Michael Tamm, who created an innovative approach that enables several types of layout bugs to be detected automatically and simply. For example, one of his tests programmatically toggles the color of the text on a page to white and then black, taking a screenshot in both cases. The difference between the two images is generated, which helps identify the text on the page. Various algorithms then detect the horizontal and vertical edges on the Web page, which typically represent elements such as text boxes and input fields. The difference of the text is then effectively superimposed on the pattern of edges to see if the text meets, or even overlaps, the edges. If so, there is a potential usability issue worth further investigation. The tests capture and annotate screenshots; this allows someone to review the potential issues quickly and decide if they are serious.

For existing tests written in WebDriver, the layout tests were enabled

by adding a couple of lines of source code. For new automated tests, some code needs to be written to navigate to the Web page to be tested before running the tests. (See <http://code.google.com/p/fighting-layout-bugs> for more information, including a video of Tamm explaining his work, sample code, and so on.)

Looking Ahead

Our work to date has been useful, and I expect to continue implementing test automation to support additional heuristics related to dynamic aspects of Web applications. WebDriver includes support for touch events and for testing on popular mobile phone platforms such as iPhone, Android, and BlackBerry. WebDriver is likely to need some additional work to support the matrix of tests across the various mobile platforms, particularly as they are frequently updated.

We are also considering writing our tests to run interactively in Web browsers; in 2009, a colleague created a proof of concept for Google's Chrome browser. This work would reduce the burden of technical knowledge to run the tests. The final area of interest is to add tests for WAI-ARIA (Web Accessibility Initiative-Accessible Rich Internet Applications; <http://www.w3.org/WAI/intro/aria.php>) and for the tests described at <http://openajax-dev.jongund.webfactional.com/rules/>.

We are actively encouraging sharing of knowledge and tools by making the work open source, and others are welcome to contribute additional tests and examples.

Conclusion

Automated testing can help catch many types of problems, especially when several techniques and approaches are used in combination. It's good to keep this in mind so we know where these automated tests fit within our overall testing approach.

With regard to the automated tests we conducted on the Google sites, the ROI for the amount of code written has justified the work. Running the tests discovered bugs that were fixed in several frontline Google properties and tools. Conservatively, the page-weight of many millions of Web re-

quests has been reduced because of problems discovered and fixed using this test automation. Keyboard navigation has also been improved for those who need or prefer using it.

Test automation is imperfect and limited, yet it can be useful in catching various problems that would trip up some of your users. The work complements other forms of testing and helps inform the project team and usability experts of potential issues quickly, cost effectively, and reliably.

Acknowledgments

Thank you to Google for allowing the original work to be open sourced, to eBay for supporting the ongoing work, to Jonas Klink for his contributions, and to various people who contributed to the article and offered ideas. Please contact the author if you are interested in contributing to the project at julianharty@gmail.com.

Steve Krugg's work is an excellent complement to automated tests. He has written two books on the topic: *Rocket Surgery Made Easy* (<http://www.sensible.com/rocketsurgery/index.html>) and *Don't Make Me Think*, of which three chapters on user testing are available to download for free from <http://www.sensible.com/secondedition/index.html>.

Related articles on queue.acm.org

Case Study: UX Design and Agile: A Natural Fit?

<http://queue.acm.org/detail.cfm?id=1891739>

Orchestrating an Automated Test Lab

Michael Donat

<http://queue.acm.org/detail.cfm?id=1046946>

Too Darned Big to Test

Keith Stobie

<http://queue.acm.org/detail.cfm?id=1046944>

Julian Harty is the tester at large at eBay, where he is working to increase the effectiveness and efficiency of testing within the organization. He is passionate about finding ways to adapt technology to work for users, rather than forcing users to adapt to (poor) technology. Much of his material is available online. He is a frequent speaker and writes about a range of topics related to technology, software testing and accessibility, among others.