GuideAutomator: Continuous Delivery of End User Documentation

Rodrigo Souza
Dept. of Computer Science
Federal University of Bahia
Salvador, Brazil
Email: rodrigorgs@ufba.br

Allan Oliveira
Dept. of Computer Science
Federal University of Bahia
Salvador, Brazil
Email: allanoliveira.main@gmail.com

***Abstract*—User guides, also known as user manuals, are a type of documentation aimed at helping a user operate a specific system. For software systems, user guides usually include screenshots that show users how to interact with the user interface. Because creating such screenshots is a slow, manual process, keeping the user guide up-to-date with changes in the user interface is challenging. We propose an approach in which the documentation writer interleaves the user guide text with source code that automates screen capturing. As a result, screenshots always reflect the latest software version, which makes the approach suitable for a project that uses continuous delivery. The approach was implemented as a prototype, called GuideAutomator.**

***Keywords*-software documentation; automated documentation generator; literate programming; continuous delivery.**

I. Introduction

The user guide for a software system is a type of documentation that shows end users the steps they should follow to accomplish specific tasks while using the software. For software systems with graphical user interfaces, user guides often include screenshots that illustrate what the users are supposed to see and how they should interact with the interface. Screenshots are sometimes annotated with arrows, rectangles or text that highlight specific elements with which the user should interact, or where the user can find important information.

In order to write the user guide for a system, a technical writer writes step-by-step instructions for the user, and then follows the instructions to take screenshots that get included in the guide. The writer may need to edit the screenshot images to crop them or to highlight important elements.

As a system evolves, keeping the documentation up-to-date is a challenge. The user interface may change, requiring the technical writer to update multiple screenshots. This is a time-intensive process because, to update a screenshot, the writer needs to follow the steps in the documentation to get the system to the state it was when the original screenshot was taken. Also, the documentation needs to be constantly reviewed to detect parts that were made obsolete by updates in the system.

Thus, keeping user guides up-to-date requires a lot of effort, ultimately slowing down the process of releasing a new version of the software. This is especially troublesome for projects with short release cycles or projects that adopt a continuous delivery approach, in which every change to the software generates a potentially shippable product.

To address those issues, we propose an approach in which the user guides are created by interleaving the document text with source code chunks that capture screenshots and insert them in the document. In this approach, the documentation can be built from its sources upon every change in the system, resulting in screenshots that always reflect the latest version of the software.

A scenario in which the approach excels is when the graphical layout of the system being documented changes. Under the traditional approach, the technical writer would have to recreate all screenshots; under our approach, the writer would simply run a command to automatically generate an updated guide with all screenshots reflecting the new layout.

This approach can also be used to reinforce acceptance testing and to detect parts of the documentation that need to be updated. For instance, if a button is removed from the graphical interface, and any documented task requires clicking on that button, the build of the documentation will stop with an error. That means that either (i) the button was essential for a use case, and the system would not satisfy the users' requirements without the button, or (ii) the steps to perform the task have changed, and the user guide should be changed to reflect that.

We implemented the proposed approach as a prototype, GuideAutomator, that accepts a user guide written in a markup language, Markdown[1], interleaved with source code chunks written in a domain-specific language for taking screenshots of web pages, GuideAutomator DSL. Since the user guide is written in plain text, it can be efficiently versioned together with the source code for the software being documented, and contributions of multiple developers can be merged using a version control system [1].

We evaluated the prototype in a pilot experiment with two programmers who were asked to write a user guide using both GuideAutomator and the traditional approach. Although the participants took longer and found it more difficult to write the user guide with GuideAutomator, they also recognized that

[1] https://daringfireball.net/projects/markdown/

our approach could take less time in the long run. Testing this hypothesis is a future work.

II. Related Work

Our approach is inspired by recent developments in reproducible research. The gold standard for reproducible research is the reporting of scientific results, including tables and figures, together with the data and source code that generated them, so other researchers can verify the findings and create derivative analyses [2]. The R Markdown package for the R programming language enables reproducible research by allowing researchers to write reports mixing Markdown text with R code chunks [3]. The report source code can be automatically *knit* into a final report, a process which replaces the R chunks with their corresponding outputs, which can be numbers, text, tables, and figures. GuideAutomator is strongly inspired by R Markdown, except that instead of running data transformation and data plotting code to generate a scientific report, it runs web and screen capturing code to generate a user guide [4].

Another inspiration is literate programming, an approach in which a program is written as a document with the program source code interleaved with text explaining how it works [5]. This document can either be transformed into pure source code, which can be compiled and executed, or can be *woven* into source code documentation. Our approach is also based on documentation interleaved with source code. In our case, however, the text does not explain the interleaved source code; instead, both the text and the source code are used to generate a user guide that explains a software system to its end users.

API documentation generators are tools that extract structured comments in the source code of a program and generate the documentation of its application programming interface (API), which is usually read by programmers. API documentation generators include Doxygen[2] and Javadoc [6]. While literate programming advocates writing explanatory text with embedded source code, the API documentation approach advocates writing source code with explanatory text embedded as source code comments. GuideAutomator is also an automated documentation generator; however, the documentation is geared towards end users, not programmers.

Although the idea of putting text and source code together is not new, our approach is novel in the way it applies this strategy to help create and maintain user guides. To the best of our knowledge, no similar approach or tool has been published.

III. Tools and Techniques

GuideAutomator is based on tools and principles from functional testing and plain text documentation, as explained next.

Functional testing is the process of checking whether the behavior of a software system conforms to its specification, from the end user point of view, by executing test cases derived from its specification [7]. For web applications, functional testing can be automated using Selenium WebDriver [8], a tool that sends commands to control a web browser and retrieves results, which can be compared to the expected results. GuideAutomator uses Selenium for a different purpose; instead of comparing the application's output with a specification, it captures the output so it can be displayed in a user manual.

Waits and Yankel [1] claim that traditional documentation tools and processes, based on the collaborative writing of documents using file formats that are binary and proprietary (e.g., Microsoft Word's `.doc`), are not well integrated into software development tools and processes. The main problems of the traditional approach are the difficulty of merging contributions of different authors, due to limitations of version control systems, and the inconsistency of document presentation across operating systems. They propose writing documentation in plain text files, using a standardized markup language such as Markdown, and then converting it into formats such as PDF and HTML using a tool such as Pandoc[3]. GuideAutomator uses both Markdown and Pandoc, and further automates the process by generating images from screen capture code interleaved with text in the documentation.

Markdown is lightweight markup language optimized for readability. It specifies text formatting of elements such as headers, bold, italic, and itemized lists using spacing and ordinary characters, such as `*`, `#`, and `-`. It is used in popular developer-centric sites and services, such as GitHub[4] and StackOverflow[5].

IV. GuideAutomator

GuideAutomator is a command-line tool that takes as input a text-only document with a specific format and creates a PDF and an HTML file representing the user guide, containing both text and screenshots. The input format mixes Markdown text with code chunks in GuideAutomator DSL, a domain-specific language based on the Selenium WebDriver API to control a web browser and take screenshots.

A. Overview

Figure 1 shows a sample input and the corresponding output. Lines 1–4 and 10–13 are written in the Markdown syntax, and appear in the output with special formatting. For instance, lines beginning with `##` become subheadings, words enclosed by `*` become *emphasized*, and words enclosed by `` ` `` become `monospaced`.

Lines 5–9 and 14–17 are code chunks written in the GuideAutomator DSL syntax. All chunks are surrounded by `<automator>` and `</automator>` tags. Inside those tags, each line is a command to a browser window that is created when the input starts to be interpreted: `get` opens a web page by its URL, `fillIn` writes text in a text box, `takeScreenshotOf` takes a screenshot of an HTML element, and `submit` submits a form. The second parameter of

[2] http://www.doxygen.org/

[3] http://pandoc.org/

[4] https://github.com/

[5] https://stackoverflow.com/

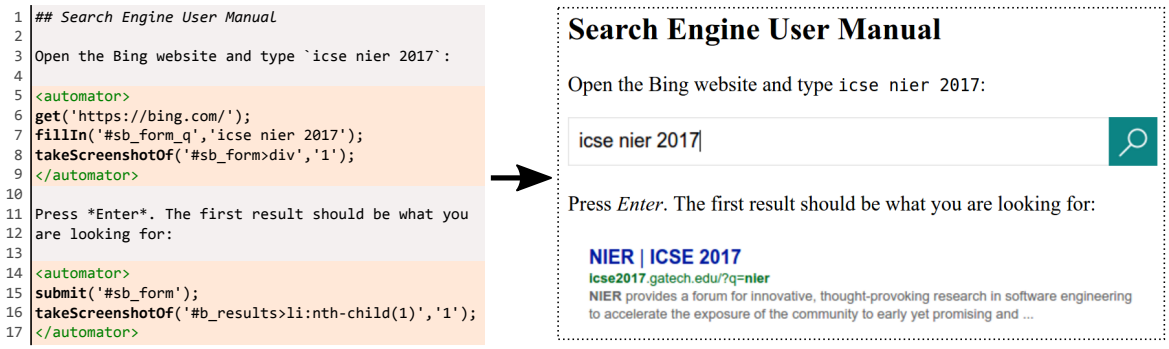

Fig. 1: Sample GuideAutomator input (left) and output (right).

`takeScreenshotOf` determines what will be captured: the whole browser viewport (`'0'`) or only the specified element (`'1'`).

The first parameter of `fillIn`, `takeScreenshotOf`, and `submit` is a CSS selector[6], which is an expression that matches an HTML element. CSS selectors are ubiquitous in web programming and design. The CSS selector on line 16, for instance, matches the first `li` (list item) element whose parent element has `b_results` as unique identifier (id). Fully explaining CSS selectors is beyond the scope of this paper.

The GuideAutomator DSL currently contains instructions for opening a URL, filling in forms, clicking on an element, surrounding elements with red rectangles for highlighting, taking screenshots, and cropping screenshots around an element. Details about all instructions can be found on GuideAutomator's website[7].

B. Usage Scenario

Currently, people who intend to write a user guide using GuideAutomator need to switch between writing the documentation text, identifying relevant URLs and web page elements, and building the documentation. This process can be performed by one person or by a multifunctional team, including, for instance, a technical writer and a web programmer.

Consider a scenario where the user guide is written sequentially by a single woman. First, she writes the initial text of the user guide, and then prepares to write the first code chunk to take a screenshot. To that end, she opens a web browser in a specific page of the software being documented and identifies key elements that should be highlighted, clicked, or filled in. Then, she writes code that references those elements by their CSS selectors. Finally, she builds the documentation and checks the output. She repeats this process until the user guide is complete.

Modern browsers' developer tools' are helpful in determining the CSS selector of an element in the page. Using such browsers, the writer can select the Inspect tool and click on the desired element. The element becomes selected and the browser generates a CSS selector that matches the element.

[6]https://www.w3.org/TR/css3-selectors/

[7]https://github.com/aside-ufba/guide-automator

The build process may have side effects that require additional precautions. If the documented tasks involve adding, removing or modifying data, the documentation should not be built against the application's production environment to avoid undesirable changes. Instead, we advise creating a dedicated application environment for building the user guide.

Also, the user guide should be reproducible: if it is built multiple times from the same documentation source and software source, the output should be the same. To achieve this effect, the software's database may need to be restored to a desirable initial state prior to every build. If the documentation is built as part of a job in a continuous integration server, the job can be configured to restore the data; otherwise, one option is to add to the application a hidden URL that, when accessed, restores the data, and then open this URL in the beginning of the first screen capture code.

V. Preliminary Evaluation

In order to get an initial assessment of GuideAutomator, we performed a pilot experiment in which two participants were asked to write part of the user guide for a medium-sized web application using two different approaches: the traditional approach and the GuideAutomator approach. For both approaches, the evaluation criteria were the time taken for writing the documentation and the subjective impression of the participants regarding the learning curve and ease of use, measured in a scale from 1 (least favorable) to 5 (most favorable). Participants were also asked about the pros and cons of each approach.

The first participant was an undergraduate Computer Science student with little web programming experience; the second one was a Computer Science professor with PhD degree and medium web programming experience. The participants were trained to create the user guide for a small application both using GuideAutomator and using traditional tools (text processors and image editors).

After the training, each participant received a PDF file containing a partial user guide built with GuideAutomator, documenting tasks for the chosen medium-sized system. The user guide contained 13 screenshots that were created by filling in forms, clicking on buttons and links, highlighting and selecting specific page elements. The participants also received

TABLE I: Experimental results for participants P1 and P2.

	Traditional	**GuideAutomator**
P1: time	41 min	71 min
P1: learning curve	5	3
P1: ease of use	5	4
P2: time	17 min	55 min
P2: learning curve	5	4
P2: ease of use	4	3

a database dump that could be used to restore the application's data to its initial state.

The participants had to write from scratch the user manual they received. More specifically, they had to do it twice: once for each approach. We measured the time participants spent writing the user guide for each approach, and collected their subjective impressions in a form.

The results of the pilot experiment are shown in Table I. Both participants spent more time writing the user guide when they used GuideAutomator. Also, they found the traditional approach easier to learn and to use. Nonetheless, they said that the tool makes it easier to create screenshots with consistent size and framing, and they recognized the tool's potential to reduce maintenance costs when the user guide needs to be updated, although this was not part of the experiment. They also enumerated some weak points of GuideAutomator, which we describe in the following section together with other challenges.

VI. Challenges

The tool that implements our approach, GuideAutomator, is still an early prototype. Here we enumerate challenges that need to be overcome to make the approach feasible for end user documentation on an industrial scale.

Usability issues. Currently GuideAutomator is a command-line tool with no graphical user interface. The technical writer has to use general-purpose developer tools embedded in the web browser to get the information needed to write the screen capturing code embedded in the user guide, which requires knowledge of web design and web programming. Also, there is a frequent context switching between text editor, command-line, and web browser, which reduces productivity. In a future work, we envision a browser plug-in that writes GuideAutomator code based on the writer's interaction with the browser.

Even for writers with programming experience, the tool lacks interactivity. Currently, the writers have to build the whole document even if they just want to check if the last code chunk works as intended. This problem can be mitigated by introducing caching and adding an interactive console where the writer can type GuideAutomator instructions that get immediately executed on GuideAutomator's browser window.

Feature limitations. Currently the output options are very limited; the tool generates HTML and PDF files with a predefined layout that cannot be customized. There is no way to create a cover page or add headers and footers. In a future work, we intend to add customization options to the tool.

When writing a user guide for a multi-language application, multiple versions of the same guide are written, one for each language. From a semantic viewpoint, all versions contain the same text and the same screenshots, only differing in the language. With our approach, all versions could share the same screen capture code, which would result in similar but distinct screenshots based on the application's language settings. Currently, however, GuideAutomator does not implement any mechanism to share code among multiple user guides.

VII. Conclusion

Keeping software systems and their documentation in sync is a challenge, especially in continuous delivery environments, in which every change to the system is potentially shippable. Updating user guides to reflect the latest changes is even harder, since it requires recreating scenarios in the system that were used to take the original screenshots.

We proposed a novel approach for creating end user documentation in which screenshots are automatically captured and are thus always in sync with the system being documented. The approach also helps putting end user documentation in the continuous integration cycle, allowing developers to identify when the documentation has diverged from the system.

We created a command-line prototype that implements the approach and evaluated it in a pilot experiment. Although participants took longer to write a user guide with GuideAutomator than using traditional tools, their feedback demonstrates that the approach is promising, especially if usability issues are fixed.

We intend to improve the tool based on the feedback received and design new experiments to better assess the potential of our approach, especially in a software evolution scenario. In one of the experiments, participants will receive the user guide of an early version of a software system and will need to update the manual to newer versions of the system, using both approaches. With this experiment we expect to get a better understanding of the time savings of GuideAutomator in the long run.

References

[1] T. Waits and J. Yankel, "Continuous system and user documentation integration," in *2014 IEEE International Professional Communication Conference (IPCC)*. IEEE, 2014, pp. 1–5.

[2] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[3] B. Baumer, M. Cetinkaya-Rundel, A. Bray, L. Loi, and N. J. Horton, "R Markdown: Integrating a reproducible analysis tool into introductory statistics," *arXiv preprint arXiv:1402.1894*, 2014.

[4] A. Oliveira and R. Souza, "GuideAutomator: Automated user manual generation with Markdown," in *Proceedings of the VII Brazilian Conf. on Software (CBSOFT), tools track*, 2016, pp. 49–56.

[5] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[6] D. Kramer, "API documentation from source code comments: a case study of javadoc," in *Proceedings of the 17th annual international conference on Computer documentation*. ACM, 1999, pp. 147–153.

[7] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[8] A. Holmes and M. Kellogg, "Automating functional tests using selenium," in *AGILE 2006 (AGILE'06)*. IEEE, 2006, pp. 6–pp.