SPIDER SENSE: Software-Engineering, Networked, System Evaluation

Nishaanth H. Reddy, Junghun Kim, Vijay Krishna Palepu, and James A. Jones
University of California, Irvine, USA
{nhreddy, junghuk, vpalepu, jajones}@uci.edu

***Abstract*—Today, many of the research innovations in software visualization and comprehension are evaluated on small-scale programs in a way that avoids actual human evaluation, despite the fact that these techniques are designed to help programmers develop and understand large and complex software. The investments required to perform such human studies often outweigh the need to publish. As such, the goal of this work (and toolkit) is to enable the evaluation of software visualizations of real-life software systems by its actual developers, as well as to understand the factors that influence adoption. The approach is to directly assist practicing software developers with visualizations through open and online collaboration tools. The mechanism by which we accomplish this goal is an online service that is linked through the projects' revision-control and build systems. We are calling this system SPIDER SENSE, and it includes web-based visualizations for software exploration that is supported by tools for mirroring development activities, automatic building and testing, and automatic instrumentation to gather dynamic-analysis data. In the future, we envision the system and toolkit to become a framework on which further visualizations and analyses are developed. SPIDER SENSE is open-source and publicly available for download and collaborative development.**

I. INTRODUCTION

The SPIDER SENSE toolkit is a platform to assist developers' comprehension of their software systems, and researchers' evaluation of software-visualization and software-comprehension tools. It is made with the intention of creating a symbiotic relationship between software-engineering (and software-visualization) researchers and software developers. Software-visualization researchers seek to enable better insight, understanding, and exploration of software subjects for tasks such as understanding architecture, finding bugs, understanding commit patterns, and assessing code coverage. Each of these example tasks can be generalized to be that of a steep learning curve problem for practicing software developers, engineers, and project managers.

However, software-visualization researchers often experience difficulty demonstrating direct impact on software-development practice. There have been some significant and notable examples of such direct impact, such as the SEESOFT visualization, which can be seen in the currently popular SUBLIME TEXT editor and development tool. However, much of software-visualization research has impacted practice through more indirect routes.

Despite the advantages of such direct application of research, researchers are often reluctant to put their research techniques and tools into practice. Research prototypes of techniques often suffer from immaturity in their implementations, which would create barriers to end-user adoption. Developers who quickly run into instabilities, crashes, or glitches are likely to abandon these tools without further time investments.

To counteract such difficulties of evaluation, researchers often utilize means of evaluation that *simulate* real-world situations. Two common ways of evaluation are to study students as a substitute for professional developers or to abstract away the human developers altogether. Studying students is particularly attractive for academics as they are readily available and can be rewarded through means such as course-assignment credit. Even further complicating this situation is that the classroom cannot replicate the organizational factors that are present in practice: for example, a hierarchy of expertise and authority, organizational policies and constraints, and customer expectations. For these reasons, program committees and journal reviewers are often critical of the ability to generalize evaluation results to software-development practice. Another method for evaluation is for the experimenter to define quantitative metrics that are meant to represent effectiveness, efficiency, or some other desired characteristic. However, these metrics are often merely proxies for real-world qualities.

The software subjects that researchers use are critical, regardless of if student experiments are performed or quantitative metrics are used. One popular and quite useful resource of software subjects is hosted at the Subject-artifact Infrastructure Repository (SIR) at the University of Nebraska, Lincoln [4]. The SIR contains many software subjects that contain faults, versions, and test cases, as well as some test and build scaffolding that facilitate experimentation. Another benefit of the subjects in the SIR is that experimenters have a *de facto* benchmark suite on which research techniques can be compared. A danger, however, is that these subjects are "fixed" — the set of faults, tests, and versions are known, and techniques can be developed that target these. Such specialization to these subjects may prove to be an eventual threat to the generalizability of results.

To overcome these issues and allow software-engineering and software-visualization researchers to evaluate the actual utility and benefit of research techniques, we have developed a toolkit that we are calling SPIDER SENSE. The SPIDER SENSE toolkit includes tools that enable a number of processes: (1) the parallel mirroring of live software development, (2) automatically generated, up-to-date visualizations of the software system under observation, (3) automatic building (*i.e.,* compiling) of the system with each new revision, (4)

automatic running of its test suite, and (5) automatic instrumentation to gather dynamic (*i.e.,* runtime) information. These research visualizations and automated-assistance tools can be made available almost immediately to the actual software developers through a web hyperlink, and as such can allow researchers to get immediate feedback on the utility of their research products.

Although today's implementation of SPIDER SENSE is complete to the point that we have been able to produce visualizations for several popular, open-source systems, we envision this framework to be the basis on which we and others continue to build further visualizations and research innovations. SPIDER SENSE is open-source and publicly download-able, and we continue to develop it to support further build environments and system-specific configurations, and enable further analyses and visualizations.

II. SPIDER SENSE

The SPIDER SENSE toolkit is composed of a number of cooperating tools, which we have developed. We created a build tool that can automatically modify a build script to enable dynamic analysis of the software subject. We also created a per-test-case instrumentation tool that allows developers, researchers, and automated analysis tools to distinguish the code coverage that was produced for each test case in a test suite. We then created a software-visualization framework that runs in a web browser, and as such enables a platform-independent visualization that can be shared directly with developers by way of a web link.

A. Build Tool

SPIDER SENSE's build tool is based on a minimally intrusive build-strategy for software systems with existing build processes. Software build processes automate day-to-day activities in software development such as, compilation, testing, analysis and deployment of software systems. As such, the scripts that support build automations for a software system are often complex and unique to the traits and characteristics of the software system being built. Even trivial changes to such complex build scripts can significantly impact the integrity of the actual software build. With such considerations, instead of changing the actual build script itself, SPIDER SENSE's build tool makes use of the final results of the software build, *i.e.,* binaries for the software system, its software-tests, and any external dependencies. Given the binaries for the software system and its tests, the build-tool runs an instrumented version of the software system, driven by the binaries of the software-tests. It is important to note that such instrumented tests are not meant to replace the execution of the actual (non-instrumented) tests of the system. In fact, to preserve the integrity of the actual build process we make special accommodations to run the instrumented tests independent of the actual tests. Such instrumented runs enable dynamic analyses of the software system which are finally used to inform various software visualizations and comprehension tools.

B. Per-Test-Case Coverage Tool

As a part of the SPIDER SENSE toolkit, we developed a simple code coverage tool called TACOCO[1] that not only provides coverage data for the software system as a whole, but also provides the exact set of source-code lines that were executed by each individual test case.

Coverage data provided by code coverage tools help software developers identify parts of their software system that were seldom or never exercised by the software system's test suite, thus indicating untested or potentially defunct parts of a software program; ultimately improving software quality. Most code-coverage tools (*e.g.,* Cobertura and JaCoCo) compute a single execution-profile (*i.e.,* the number of times a set of software source-code lines were executed) for the entire software system from the beginning to the end of an entire test-suite's execution. As a consequence, code coverage is often computed by treating the software system and its test suite as monolithic entities.

However, different test-cases in a test suite are often designed to test varying aspects of the software system. For instance, unit tests are written to test individual modules of the software system, in isolation from the rest of the system. Integration tests are designed to test the various ways by which different modules of the system interact with each other. And system tests verify the system wide workings of the software. As such, individual tests not only exercise and verify unique parts, but also unique facets of the software system. Thus, by knowing the exact parts covered by each such individual test-case, SPIDER SENSE can provide insight into the purpose of the individual test-cases, and also highlight the various facets of the software system.

Moreover, several automated software-engineering research innovations depend upon per-test-case coverage information, such as spectra-based fault localization (*e.g.,* TARANTULA [7]) and regression test selection (*e.g.,* DEJAVU [9]). In the past, researchers have achieved per-test-case coverage by running the instrumented program with each input as if it were the entire test suite, then saving and resetting the coverage state for the next input. However, with the JUNIT test framework, individual test cases are contained with methods, further contained within test-case classes. As such, starting and stopping instrumentation for individual methods is not as straightforward, particularly when considering shared set-up and tear-down routines and other advanced testing mechanisms, such as parameterized test runners.

TACOCO computes the per-test-case coverage by maintaining an individual execution profile for each individual test-case-method execution. Our current implementation does this specifically for the JUNIT test-framework, where the execution of a test-method inside a test-class is treated as an individual test-case execution. TACOCO tracks the moments in time when the execution of a test-method begins and the moments in time

[1] TACOCO is developed and maintained as an open-source project on Github: https://github.com/spideruci/tacoco. Along with the implementation for the per-test-case coverage tool, TACOCO also comes with an implementation of the build-tool that was described in Section II-A.

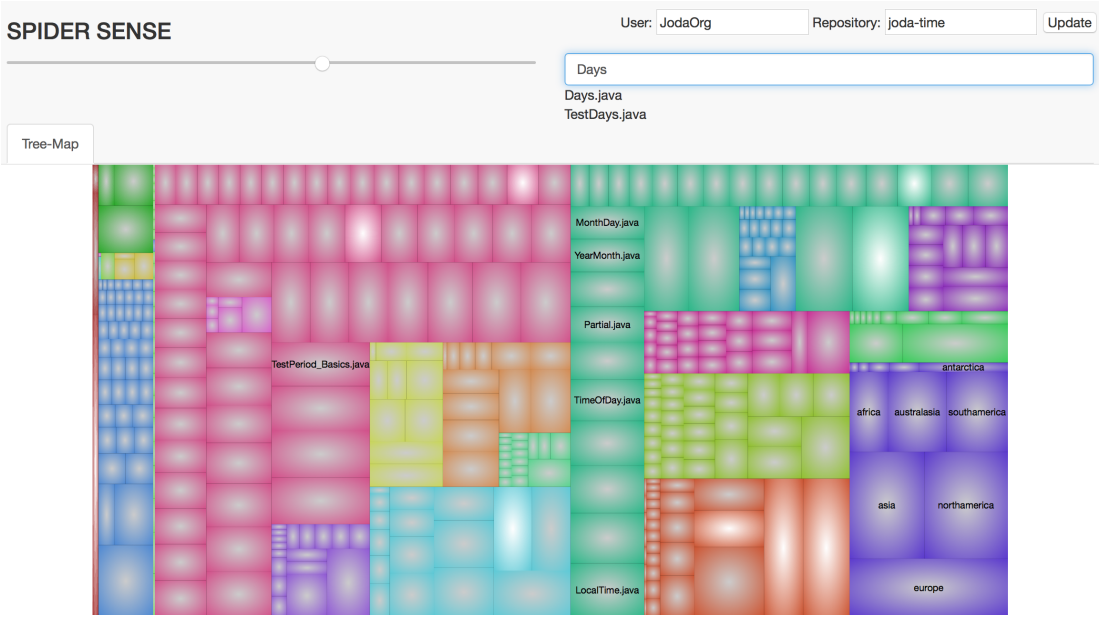

Fig. 1: Treemap view in SPIDER SENSE of the package hierarchy of the JODATIME Java library's main source code.

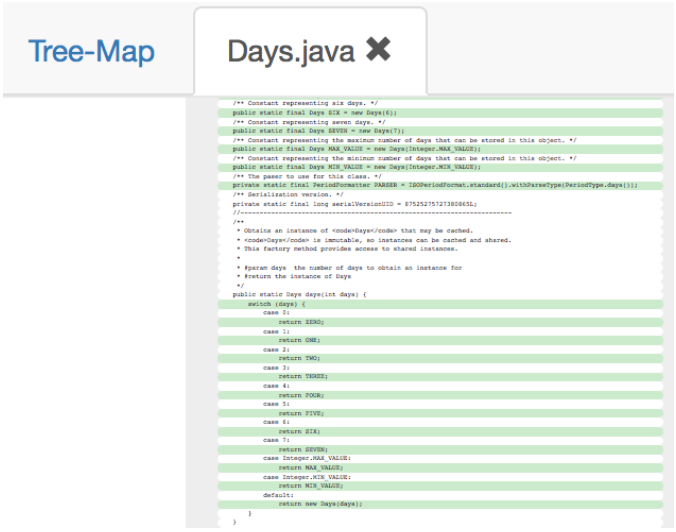

Fig. 2: Seesoft view in SPIDER SENSE of the `Days.java` source file, with lines executed by passing tests colored in green, with the Tarantula suspiciousness metric.

that the executions returns back from a test-method during a JUNIT test-suite execution. Using this timing information, provided through *event listeners*, at the beginning of a test-method's invocation, SPIDER SENSE creates a fresh instance of the data-structure that stores the execution profile that indicates that the test-method has executed no source-code so far. And just before a test-method is about to return, the current state of the execution-profile for the test-method is saved in a dictionary, with the test-method as the key. This enables the storage of execution-profiles and finally the coverage information, on a per-test-case basis. Moreover, this approach, which is based on the dynamic observations of execution, can properly accommodate parameterized test runners in which the same test-method is called repeatedly with different "parameters."

C. Web-Based Visualizations

The web-based visualization[2] presented in the SPIDER SENSE toolkit is comprised of many components controlled by event listeners using the D3.JS library [2] . The system makes asynchronous web requests whenever a new view is opened. The system can be broadly broken down into the following parts:

1) Navigation Bar: The web-based visualizations in SPIDER SENSE are guided with a navigation bar, as shown in Figure 1, at the top of the visualizations with the functionality needed to carry out all the view switching, zooming and searching that is required. The navigation bar is intuitively designed to make space for tabs containing different visualizations. The zoom bar is linked to the currently opened view. The search bar is designed to auto-complete search queries and will open the file that is searched in a new tab, as shown in Figure 1. The tab system is designed to support independent canvases for any general purpose visualization and as such, serves as a point of extension for integrating new visualizations and tools into SPIDER SENSE.

2) Treemap: Upon providing the details about the GitHub repository and username, a web API request is made to obtain the details about the file structure, file sizes and recent commit information for the corresponding software system. These values are used to create a treemap view of the files present in the GitHub repository, using the D3.JS visualization

[2]The web-based visualization is maintained in a public git repository called SENSE VIS:https://github.com/spideruci/sense-vis

framework. Each rectangle represents a source file or package, and the size of the rectangle is proportional to the number of lines of code in that represented entity. Secondary information, such as the recency of file commits, is configured into the visualization using the brightness of the rectangular elements in the treemap. On clicking any rectangle, one can move a folder down the hierarchy, and this is captured in the treemap by means of transitions and zooming. A right-click opens the file in a new tab in the Seesoft view (Section II-C3). On opening a Seesoft view, the treemap view is hidden until the treemap tab is clicked.

3) Seesoft: The Seesoft view, as shown in Figure 2, displays the source code in a "zoomed-out" fashion, and it colors the lines of code in a parameterizable way so as to allow for different visualizations and applications. Every line of code is made interactive and zoomable by using event listeners. When zoomed out, clicking on any line of code would zoom in and auto-scroll the line of code selected. A double-click redirects the user to the GitHub page with the line of source code highlighted. As a sample application of the Seesoft visualization we compute the suspiciousness values, using the Tarantula metric [7] for fault localization, for every line of code with respect to the per test case code coverage generated with TACOCO.

Figure 2 shows the Seesoft view of the `Days.java`, while highlighting the lines executed with passing test cases. Note that at the time of testing SPIDER SENSE all tests for JODATIME were passing, and as such we found that all executed lines were colored with green. Further, as a consequence of augmenting the Seesoft view with runtime data, non-executable source-lines such as comments and method declarations were left colorless, making them distinctly visible against the executable lines of code. Such examples of immediate feedback and insight about real world projects were possible when data about the software system was presented as a visualization.

III. DEMONSTRATION

We will now present a brief walk-through of the SPIDER SENSE toolkit using the software system JODATIME as a concrete example of a software system that we need to setup for analysis. The purpose of this walk-through is to demonstrate the basic usage of SPIDER SENSE. There are three basic steps involved in setting up any software subject for analysis, either by a software researcher or a software developer who is trying to better understand the software subject: (a) building the software subject; (b) gathering runtime data about the software; (c) and finally visualizing the runtime data to better comprehend the software. We will start with building the software subject.

A. Building the Software Subject

To apply SPIDER SENSE to the new software subject, JODATIME, we first obtain a copy of JODATIME by simple cloning a public version of its `git` repository hosted at GitHub.[3] Upon obtaining the repository, we notice that JODATIME's build system uses maven, and so we simply build the project with the command `mvn test`, which compiles the main-system- and test-source-code for JODATIME into executable binaries and runs all available unit tests. Running the actual, non-instrumented tests allows us to verify that the system actually builds correctly. Once we have verified that the system actually builds successfully without any instrumentation, we use TACOCO to carry out an instrumented execution of the test-cases using the binaries for the actual JODATIME library and its test-suite.

B. Gathering Runtime Analysis Information

Next, we use TACOCO to get per-test-case code coverage information. We first obtain TACOCO from its public repository[4] and build TACOCO using MAVEN, *i.e.,* we execute the command `mvn test`, just like we did with JODATIME. Next, we will run the instrumented version of JODATIME, by using a shell script command called `run-tacoco` that takes two inputs: the paths to the software builds of JODATIME and TACOCO. The script can capture all of the library dependencies, build configurations, and directory structures that are necessary to build, test, and instrument the program by executing a series of MAVEN commands. In this way, the build script does not need to be modified in any way, but can still benefit from the automation provided from it. The execution of this script will result in an instrumented run of JODATIME's test cases, and it will run all of the test cases that `mvn test` ran, to finally produce the per-test-case code coverage data file for all tests. This code coverage data file can be converted into a suitable JSON file format to be consumed by the web-based visual interface, in our final step in setting up JODATIME.

C. Displaying Software Visualizations

Once we convert the code-coverage data file into a suitable JSON format, the files are uploaded a location that is referenced by the web-based visualization. The SPIDER SENSE visualization then uses this coverage information to color the source-code lines for individual files as shown in Figure 2. While the build tool provides us with the coverage data, the web-based visualization is capable of independently obtaining the project's source-code data from GitHub and displaying it as a Treemap view, as shown in Figure 1.

Once this setup is complete, the user (researcher or developer) can open the SPIDER SENSE dashboard and load the required software project by providing details of its Github repository. The user can then navigate to the file of interest by simply performing a search as shown in Figure 1, where the results for the querry "Days" are presented. On opening the Seesoft view for a given file (*e.g.,* `Days.java`) the user would be able to see the lines of code colored by their suspiciousness values. The user would also potentially be able to open the Seesoft views for other files and investigate the suspiciousness values across different source files. This would

[3] JODATIME is hosted at https://github.com/JodaOrg/joda-time

[4] https://github.com/spideruci/tacoco

provide the user the required information to carry out a basic fault localization task, given a set of failing tests.

IV. Related Works

Eick *et al.* developed the Seesoft visualization [5] that we use in displaying individual source file. Our implementation is, however, zoomable to enable exploration of the details in context. Moreover, our implementation is web-based, and as such allows users to view the visualizations on any platform capable of loading HTML5 web content.

Blackburn *et al.* developed the DaCapo benchmarking system [1] to provide a platform on top of which a variety of Java project codebases can be tested. Do *et al.* developed the Software-artifact Infrastucture Repository (SIR) [4] which served as a platform for controlled experimentation with testing and regression testing techniques. DaCapo and SIR both provide software-subject evaluation platforms but provide fixed software subjects, as opposed to a framework to easily add new, live, currently-in-development software subjects and versions.

Dallmeier *et al.* presented the iBugs [3] platform that used the AMPLE bug-localization tool to extract benchmarks for bug localization from the history of a project. Although the vision of iBugs was to support automatic analysis of new software subjects, that vision was never achieved in practice — it supports exactly three hard-coded software subjects (AspectJ, Rhino, and *partially* JodaTime). Gousios created the GHTorrent Dataset and Tool Suite [6] that provided a REST API that in turn used the GitHub API asynchronously to collect data from existing GitHub repositories. This is similar to our work in the sense that it provides opportunities for the research community, however it does not provide the same build, test, instrumentation, and visualization facilities that Spider SENSE provides.

Continuous integration systems such as Travis and Jenkins provide a platforms to build and test existing projects. But these systems provide simple analytics about passing or failing builds and test cases. Coveralls[5] and Codecov[6] are platforms to check the code coverage of test cases on repositories. These tools also provide a view of every file showing whether or not lines of code have been covered. They do not, however, provide per-test-case coverage information that is needed for many software-engineering research techniques, nor do they provide extensible facilities for developing research techniques and visualizations.

The Gammatella system [8] developed by Jones *et al.* uses treemaps and a Seesoft view. In contrast, Spider SENSE is built to support easy adoption of new software subjects in an up-to-date fashion, and support extensibility to future visualizations. However, one perspective that can be seen is that Spider SENSE is an extension of the original Gammatella work.

[5]https://coveralls.io

[6]https://codecov.io

V. Conclusions

In this paper, we presented a toolkit for mirroring actual, live, real-world software development, which can then provide the actual software developers with up-to-date software visualizations for program comprehension. The toolkit, Spider SENSE, provides a web-based interface that supports a guided exploration of GitHub repositories. We have implemented a set of tools that comprise Spider SENSE that automate or assist its application onto new software subjects, with minimal effort by researchers or developers. Spider SENSE is open-source and publicly downloadable to enable open development and collaboration with other researchers and developers for ours and additional software-visualization and software-engineering research innovations.

VI. Acknowledgements

We would like to acknowledge the software-development efforts of Lawrence Yu in the early versions of the per-test-case code-coverage tool, and of Bixia Si, Max Wei, and Anshuman Sanghvi for their contributions to the software development of Spider SENSE. This work is supported by the National Science Foundation under awards CAREER CCF-1350837 and CCF-1116943.

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *oopsla*, pages 169–190, 2006.

[2] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[3] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM.

[4] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 60–70, Washington, DC, USA, 2004. IEEE Computer Society.

[5] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11), 1992.

[6] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236. IEEE Press, 2013.

[7] J. A. Jones. Semi-automatic fault localization. 2008.

[8] J. A. Jones, A. Orso, and M. J. Harrold. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[9] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering, IEEE Transactions on*, 24(6):401 –419, jun 1998.