You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering

Philip Johnson

Department of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822
(808) 956-3489
(808) 956-3548 (fax)
johnson@hawaii.edu

October 31, 2001

Abstract

Collection and analysis of empirical software project data is central to modern techniques for improving software quality, programmer productivity, and the economics of software project development. Unfortunately, barriers surrounding the cost, quality, and utility of empirical project data hamper effective collection and application in many software development organizations.

This paper describes Hackystat, an approach to enabling ubiquitous collection and analysis of empirical software project data. The approach rests on three design criteria: data collection and analysis must be developer-centric rather than management-centric; it must be in-process rather than between-process, and it must be non-disruptive—it must not require developers to interrupt their activities to collect and/or analyze data. Hackystat is being implemented via an open source, sensor and web service based architecture. After a developer instruments their commercial development environment tools (such as their compiler, editor, version control system, and so forth) with Hackystat sensors, data is silently and unobtrusively collected and sent to a centralized web service. The web service runs analysis mechanisms over the data and sends email notifications back to a developer when "interesting" changes in their process or product occur.

Our research so far has yielded an initial operational release in daily use with a small set of sensors and analysis mechanisms, and a research agenda for expansion in the tools, the sensor data types, and the analyses. Our research has also identified several critical technical and social barriers, including: the fidelity of the sensors; the coverage of the sensors; the APIs exposed by commercial tools for instrumentation; and the security and privacy considerations required to avoid adoption problems due to the spectre of "Big Brother".

Overview

Collection and analysis of empirical software project data is central to modern techniques for improving software quality, programmer productivity, and the economics of software project development. Unfortunately, effective collection and analysis of software project data is rare in mainstream software development. Prior research suggests that three primary barriers are: (1) *cost*: gathering empirical software engineering project data is frequently expensive in resources and time; (2) *quality*: it is often difficult to validate the accuracy of the data; and (3) *utility*: many metrics programs succeed in collecting data but fail to make that data useful to developers. Removing these barriers to widespread adoption of empirically-based techniques is an important goal for future software engineering research.

We have recently initiated a long-term research project called Hackystat, which explores the strengths and weaknesses of a *developer-centric*, *in-process*, and *non-disruptive* approach to addressing these barriers to widespread adoption of empirical software project data collection and analysis. Hackystat makes available to developers a set of custom sensors that they voluntarily attach to development tools such as their compiler, editor, configuration management system, testing framework, and so forth. Once installed, these sensors automatically monitor characteristics of the developer's process and products and send the resulting data to a centralized web service. The web service maintains a personal repository of empirical software engineering data for each developer, performs analyses on the raw data stream, and automatically sends the developer an email when new, unexpected, and/or potentially interesting analysis results become available.

Our long-range goal is to provide an "early warning system" for developers that improves awareness of potential design or implementation problems in development, and that enables "just-in-time" coordination among groups of developers on a project. For example, the "test first design" technique in Extreme Programming leads to a mixture of passing and failing tests throughout development, so developers become habituated to the presence of failures during testing. However, some failures are more "interesting" than others: the failure of a test case regarding one module due to a change in another module might indicate the need for refactoring; it might also indicate the need to consult with a developer of that other module. Identifying the correct developer to contact might require information regarding who made what changes over time, and how those changes correlated with past test case success and failure. This single scenario requires the synthesis and analysis of data gathered by multiple sensors, including a sensor for the unit testing framework (to detect the test pattern), a sensor for the software structure (to assess the potential for refactoring through analyses such as the Chidamber-Kemerer metrics), and a sensor for the configuration management system (to detect the correct developer to contact).

Hackystat is developer-centric because data is collected directly from developer activities, and analyses are provided back to developers as opposed to their management-level superiors. It is in-process because data is both collected and analyzed regularly throughout project development. It is non-disruptive because developers do not need to interact with the sensors once installed and configured.

Our pursuit of a developer-centric, in-process, and non-disruptive approach to software project data collection and analysis stems to some extent from our successes over the past ten years in empirical software engineering research. It stems to a greater extent from our failures. In the CSRS project, we built a computer-supported cooperative work environment to support software review [1, 2, 3, 10]. CSRS included a hypertext database engine, a review process modelling language, and fine-grained instrumentation support. CSRS provided excellent infrastructure for controlled experimentation on software review [7, 9]. However, although CSRS was evaluated by several industrial organizations, developers found the process model to be overly restrictive and did not want to provide managers with such detailed information concerning their development activities. CSRS thus failed to be sufficiently developer-centric.

In the Leap project, we built a toolkit for use by individual developers to record time, size, and defect data to support quality assurance and project planning [4, 8]. Leap was designed to overcome quality and overhead issues that we previously discovered in research on the Personal Software Process [5]. The toolkit

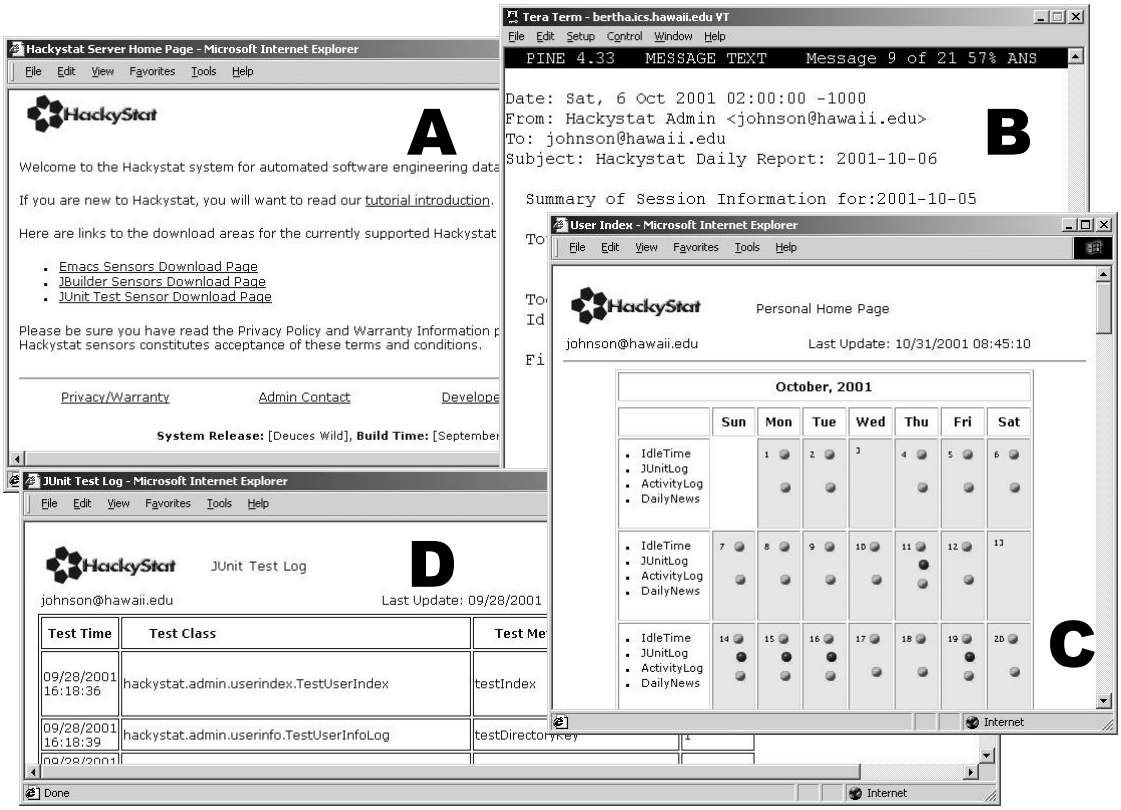

Figure 1: *Sample screenshots illustrating the current Hackystat implementation.*

not only improved the quality of data, but also led to interesting insights into the relative effectiveness of the PROBE estimation method relative to other, simpler approaches [6]. To lower overhead, Leap provided easy-to-use, GUI-based tools to enter personal project data and perform a wide range of analyses on the historical database built by the developer over time. To our dismay, we discovered that for many developers, including ourselves, even pushing a button could sometimes constitute "too much overhead". In addition, the analyses provided by Leap do not begin to be of use until after weeks or months of data collection, and the effort required during that time constitutes a large leap of faith for many developers. This system thus failed to be sufficiently non-disruptive.

While a developer-centric, in-process, and non-disruptive approach thus appears to address many of the problems preventing ubiquitous empirically-guided software engineering, it also creates its own set of new critical technical barriers. While automated measurement clearly lowers the cost of data collection, is the data collected in this way sufficiently accurate for the purposes to which it must be applied? Will commercial tools expose the appropriate API to support the kinds of instrumentation needed? Certain kinds of project data can't be sensed automatically—what limitations does this place on the utility of the system? Finally, will our envisioned privacy and security mechanisms suffice to prevent adoption problems due to the spectre of "Big Brother"?

The reference implementation

We began work on a reference implementation of Hackystat in May, 2001. Figure 1 illustrates some of the features of the current Hackystat implementation. Developers use the web server to download and install sensors for a variety of tools, as shown in Screen A. Once installed, the sensors send data gathered from tool

usage to the server, which analyzes it and sends email as shown in Screen B back to the developer whenever analyses indicate important or anomalous trends, but no more than once a day. The email contains URLs which can be used to "drill down" into the data repository if the developer so desires. Screen C provides an overview of the collected data, with button links to individual log files such as the one shown in Screen D. A button link is green if the data inside appears "normal", and red if the data inside contains anomalies or other information that should be brought to the attention of the developer.

One approach to identifying certain kinds of important changes or anomalies in the data stream is statistical process control. For example, appropriate control charts can enable the system under certain circumstances to detect when recent data becomes "significantly different" from past data.

The current system implements only a small set of sensors for developer activities within an IDE (such as compilation, visiting a file, or saving a file), the elapsed time and idle time within an IDE, and the results of running JUnit tests through a custom version of the junit.jar file. However, we have identified over a dozen sensor data types that can detect over two dozen potentially anomalous conditions.

Short-range research

Over the next three to five years, we intend to explore this approach through the following research activities:

1. *Bootstrap and ongoing technology development.* The current system does not have a sufficient number of sensors and analysis mechanisms. The "bootstrap" phase will create a critical mass of sensors and analysis mechanisms required for experimentation.

2. *Verification and validation.* Verification activities will assess the fidelity of the sensors: can they record data with sufficient accuracy without developer intervention? Validation activities will assess the utility of the analyses: do developers find the analyses to be useful, and do they actually make changes based upon the feedback they receive?

3. *A comparative study of data collection and analysis in Hackystat and the PSP*. The Personal Software Process (PSP) is a developer-centric, in-process, *disruptive* approach to software project data collection and analysis. We will perform a case study to compare the strengths and weaknesses of these two approaches.

4. *A case study of automated data collection and analysis for Extreme Programming.* This case study will explore whether the Hackystat approach can add value and provide new insight into "agile" development methods such as XP.

5. *A longitudinal study of software development skill maturation.* We will deploy Hackystat into the lab environment at the University of Hawaii in the next year. By three years from now, we will have in-process software development data from students over two years of course work. This study will provide insights into the development of advanced programmers, with the goal of improving educational practice.

Long-range research

Our short-range research objectives should provide concrete results regarding the viability of the approach and comparisons to other software engineering techniques. We have also identified several longer range research objectives:

1. *User modeling of software engineers.* The data collected by Hackystat constitutes a "trace" of developer activities over time and their impact upon a range of software products. Can user modeling techniques create a more robust model of developers that can provide better insight into their strengths and weaknesses?

2. *Collaborative process modeling of software development groups.* Not only does data collected by Hackystat constitute a trace of individual developers, it also implicitly constitutes a trace of the group process. Can we apply grammar-based approaches to infer positive and negative collaborative processes and have the environment suggest appropriate corrective action?

3. *Reasoning over the web of Hackystat servers.* The Semantic Web project is an initiative to make data on the web more easily repurposed and processable by machines on a global scale. A similar approach could be applied to the web of Hackystat servers. Potential applications could include querying other servers for "similar" process/product situations to improve local analyses, and higher-level descriptions of software development dynamics at the regional or global level.

References

[1] Philip M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering*, May 1994.

[2] Philip M. Johnson. Design for instrumentation: High quality measurement of formal technical review. *Software Quality Journal*, 5(3):33–51, March 1996.

[3] Philip M. Johnson. Reengineering inspection: The future of formal technical review. *Communications of the ACM*, 41(2):49–52, February 1998.

[4] Philip M. Johnson. Leap: A "personal information environment" for software engineers. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, CA., May 1999.

[5] Philip M. Johnson and Anne M. Disney. A critical analysis of PSP data quality: Results from a case study. *Journal of Empirical Software Engineering*, December 1999.

[6] Philip M. Johnson, Carleton A. Moore, Joseph A. Dane, and Robert S. Brewer. Empirically guided software effort guesstimation. *IEEE Software*, 17(6), December 2000.

[7] Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Journal of Empirical Software Engineering*, 4(1):9–35, January 1998.

[8] Carleton A. Moore. *Investigating Individual Software Development: An Evaluation of the Leap Toolkit*. Ph.D. thesis, University of Hawaii, Department of Information and Computer Sciences, August 2000.

[9] Adam A. Porter and Philip M. Johnson. Assessing software review meetings: Results of a comparative analysis of two experimental studies. *IEEE Transactions on Software Engineering*, 23(3):129–145, March 1997.

[10] Danu Tjahjono. *Exploring the effectiveness of formal technical review factors with CSRS, a collaborative software review system*. Ph.D. thesis, Department of Information and Computer Sciences, University of Hawaii, August 1996.