Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined

Philip M. Johnson Hongbing Kou Joy Agustin Christopher Chan
Carleton Moore Jitender Miglani Shenyan Zhen William E.J. Doane
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu

Abstract

Pedagogies such as the Personal Software Process (PSP) shift metrics definition, collection, and analysis from the organizational level to the individual level. While case study research indicates that the PSP can provide software engineering students with empirical support for improving estimation and quality assurance, there is little evidence that many students continue to use the PSP when no longer required to do so. Our research suggests that this "PSP adoption problem" may be due to two problems: the high overhead of PSP-style metrics collection and analysis, and the requirement that PSP users "context switch" between product development and process recording. This paper overviews our initial PSP experiences, our first attempt to solve the PSP adoption problem with the LEAP system, and our current approach called Hackystat. This approach fully automates both data collection and analysis, which eliminates overhead and context switching. However, Hackystat changes the kind of metrics data that is collected, and introduces new privacy-related adoption issues of its own.

1. Introduction

> It would be nice to have a tool to automatically gather the PSP data. Because judgment is involved in most personal process data, however, no such tool exists or is likely in the near future.
> —*A Discipline for Software Engineering*[4]

Until the mid-1990's, most software engineering metrics were designed for use at the organizational rather than the individual level. A best practice for organizational metrics programs is the Software Process Group, which takes responsibility for collection and analysis of metric data. Indeed, an important reason for these groups is to prevent the failure of the metrics program due to increased developer workload. For example, an industrial case study of code inspection found that adoption was facilitated by providing additional staff for metrics collection and analysis in order to "overcome [developer's] natural resistance to paperwork—a syndrome typical of most metrics programs" [12].

In 1995, Watts Humphrey authored *A Discipline for Software Engineering*, a ground-breaking text that adapted organizational-level software measurement and analysis techniques to the individual developer along with a one semester curriculum. These techniques are called the Personal Software Process (PSP)[1]. The primary goals of the PSP are to improve project estimation and quality assurance. These goals are pursued by collecting size, time, and defect data on an initial set of software projects and performing various analyses on it. For example, given the estimated size of a new system, a PSP analysis called PROBE provides an estimate of the time required based upon the relationship between time and size on prior projects.

The PSP incorporates two interesting conjectures. First, it conjectures that PSP-style collection and analysis of metric data by an individual can provide significant benefits to that individual. Second, it conjectures that these benefits are large enough that the developer will continue to use the PSP after leaving the classroom.

Over the past six years, case studies have tested the first conjecture by analysis of the student-collected PSP data [7, 9, 10, 5, 13]. These studies support the first conjecture,

[1]Personal Software Process and PSP are registered service marks of Carnegie Mellon University

Characteristic	**Generation 1** (manual PSP)	**Generation 2** (Leap, PSP Studio, PSP Dashboard)	**Generation 3** (Hackystat)
Collection overhead	High	Medium	None
Analysis overhead	High	Low	None
Context switching	Yes	Yes	No
Metrics changes	Simple	Software edits	Tool dependent
Adoption barriers	Overhead, Context-switching	Context-switching	Privacy, Sensor availability

Figure 1. Three generations of approaches to metrics for individuals

indicating that PSP data can support both project estimation and quality assurance. In addition, researchers from the Software Engineering Institute analyzed data submitted to them by instructors of 23 PSP classes, and concluded that the PSP improved the students' estimation accuracy and product quality [2].

To our knowledge, there is no published empirical research directly addressing the second conjecture, such as studies reporting the actual percentage of PSP students who continue to use it one, two, and three years after having taken the class. (Publications on PSP adoption generally consist of speculative guidelines and/or short-term data which do not address the second conjecture.) However, anecdotal evidence does not support the second conjecture. For example, a report on a workshop of PSP instructors reveals that in one course of 78 students, 72 of them "abandoned" the PSP because they felt "it would impose an excessively strict process on them and that the extra work would not pay off." None of the remaining six students reported any perceived process improvements [1]. Our experiences teaching the PSP are similar: despite classroom improvements in estimation and quality assurance, few if any students adopted PSP-specific concepts.

If the first conjecture is true but the second is false, then studying the PSP is similar to studying Latin: a task that advocates suggest you learn for its indirect benefits, rather than because you'll actually use it in your daily life. The workshop report echos this PSP-as-Latin viewpoint when it conjectures, "Even if students don't use the PSP again, improving and making them aware of their programming habits will help them in their future academic and professional careers."

This paper presents a perspective on our research and educational experiences for the past six years regarding metrics collection and analysis for individual developers. We began by teaching and using the PSP in its original form, but students found the overhead of metrics collection and analysis to be excessive. To address this issue, we next developed a comprehensive toolkit for PSP-style metrics collection and analysis called Leap [8, 6]. Despite the automated support, adoption was still low, and this led us to the discovery of another adoption barrier: the need for students to continuously "context switch" between product development and process recording. We have now implemented a new system called Hackystat and have deployed it in two software engineering classes. Hackystat completely automates both collection and analysis of metric data and thus addresses both the overhead and context switching barriers to adoption. The next section discusses this research trajectory in more detail.

2. Three generations of metrics for individuals

Looking back, we can divide our research on metrics for individuals into three generations. Figure 1 illustrates five distinguishing characteristics of these generations.

The first generation approach uses the PSP as originally described in *A Discipline for Software Engineering.* Users of the PSP create and print out forms in which they manually log effort, size, and defect information. Additional forms use this data to support project estimation and quality assurance. This approach creates substantial overhead due to form filling. For example, the PSP requires students to write down every compiler error that occurs during development. It also recommends that the developer keep a stopwatch by their desk in order to keep track of all interruptions. A benefit of using forms is that changing metrics simply involves editing the affected forms and/or creating new ones.

We began teaching the PSP in 1996, and had success similar to that reported in other case studies. Most of our students were able to estimate both the size and time of their final project with 90% or better accuracy, and one student achieved 100% yield on one project. (This means that the student eliminated all syntax and semantic errors from the system prior to the first compile of that system.) Despite the obvious discipline displayed by these students, followup

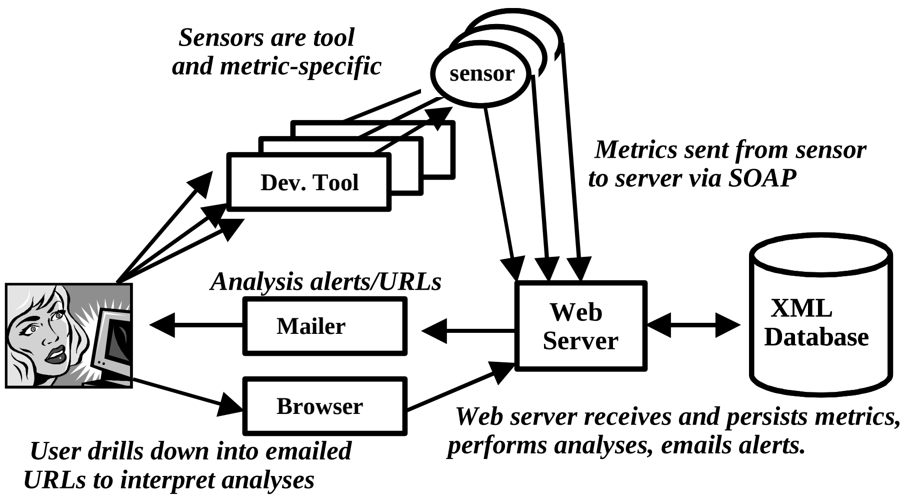

Sensors are tool and metric-specific
sensor
Dev. Tool
Metrics sent from sensor to server via SOAP
Analysis alerts/URLs
Mailer
Web Server
XML Database
Browser
Web server receives and persists metrics, performs analyses, emails alerts.
User drills down into emailed URLs to interpret analyses

Figure 2. The basic Hackystat architecture and information flow

email indicated that none of them continued using the manual PSP after finishing the semester. We attributed this to the overhead involved in collection and analysis, and began the Leap research project in 1998 to pursue low overhead approaches to collection and analysis of individual software engineering metric data.

A second generation approach uses Leap or another automated tool for PSP-style metrics such as the PSP Studio [3] and PSP Dashboard [11]. These tools all have the same basic approach to user interaction: they display dialog boxes where the user records effort, size, and defect information. The tools also display various analyses when requested by the user. Second generation approaches do an excellent job of lowering the overhead associated with metrics analysis, and substantially reduce the overhead of metrics collection. However, metrics changes require changes to the software and are thus more complicated than in the first generation approach.

After teaching and using the Leap system we found that, similar to the manual PSP, developers can utilize their Leap historical data to substantially improve their project planning and quality assurance activities. Followup email indicated that adoption improved slightly: a handful of students continued to use Leap after the end of the semester, and a small number of industrial developers discovered the tool online and began using it. A few former students and developers continue to use at least some parts of Leap.

While “some adoption” is definitely an improvement over “no adoption”, we were still surprised by the very low level of adoption of a toolkit that provided so much automated support. We then discovered that a major adoption barrier is the requirement that the user constantly switch back and forth between doing work and “telling the tool” what work is being done. Even if telling the tool is as simple and fast as pressing a button, this continual context switching is still too intrusive for many users who desire long periods of uninterrupted focus for efficient and effective development.

In May 2001, we began the Hackystat project, in which metrics are collected automatically by attaching sensors to development tools, metric data is sent by the sensors to a server, analyses over the gathered data are performed by a server, and alerts are emailed to the developer when triggered. With Hackystat, the overhead of metrics collection is effectively eliminated, developers never context switch between working and telling the tool that they’re working, and analysis results can be provided in a “just in time” manner. While Hackystat successfully addresses the barriers to adoption identified in first and second generation approaches, it changes the nature of metric data that is collected, imposes requirements on development tools, and introduces new adoption issues. The remainder of the paper describes the system and our results in more detail.

3. An overview of Hackystat

Figure 2 shows the basic architecture of Hackystat and how information flows between the user and the system. Hackystat requires the development of client-side sensors that attach to development tools and that unobtrusively collect effort, size, defect, and other metrics regarding the user’s development activities. Not every development tool is amenable to Hackystat instrumentation: Emacs is easy to integrate, Notepad is not.

The current system includes sensors for the Emacs and JBuilder IDEs, the Ant build system, and the JUnit testing tool. These sensors collect activity data (such as which file, if any, is under active modification by the developer at 30 second intervals), size data (such as the Chidamber-Kemerer object oriented metrics and non-comment source lines of Java code), and defect data (invocation of unit tests and their pass/fail status).

The developer begins using Hackystat by installing one or more sensors, and registering with a Hackystat server. During registration, the server sets up an account for the developer and sends her an email containing a randomly generated 12 character key that serves as her account password. This password prevents others from accessing her metric data or uploading their data to her account.

Once the developer has registered with a server and installed the sensors, she can return to her development activities. Metrics are collected by the sensors and sent unobtrusively to the server at regular intervals (if the developer is connected to the net) or cached locally for later sending (if the developer is working off line).

On the server side, analysis programs are run regularly over all of the metrics for each developer. A fundamental analysis is the abstraction of the raw metric data stream into a representation of the developer's activity at 5 minute intervals over the course of a day. We call this abstraction the "Daily Diary", and it is illustrated in Figure 3. This Daily Diary shows that the developer began work on Friday, June 21, 2002, at approximately 9:30am, and during the first five minutes of work the file that was edited most frequently was called Controller.java. The location of this file is also indicated along with its Chidamber-Kemerer metrics and size, computed from the .class file associated with the most recent compilation of this file in the developer workspace. Among other things, this Diary excerpt also shows that between 9:45am and 9:55am, the developer invoked 60 JUnit tests that passed, 1 that failed, and none that aborted due to exceptional conditions.

The Daily Diary is useful for visualizing and explaining Hackystat's representation of developer behavior, but is not intended as the "user interface" to the system. Instead, the Daily Diary representation serves as a basis for generating other analyses, such as: the amount of developer effort spent on a given module per day (or week, or month); the change in size of a module per day (or week, or month); the distribution of unit tests across a module, their invocation rate, and their success rate per day (or week, or month), the average number of new classes, methods, or lines of code written in a given module per day (or week, or month), and so forth.

Analysis results are available to each developer from their account home page on the web server, and can be retrieved manually to support, for example, project planning activities. However, a more interesting mechanism in Hackystat is the ability to define alerts, which are analyses that run periodically over developer data and that specify some sort of threshold value for the analysis. If the threshold is exceeded, the server sends an email to the developer indicating that an analysis has discovered data that may be of interest to the developer along with an URL to display more details about the data in question at the server.

One alert is called the "Complexity Threshold Alert", and it allows the developer to configure it to analyze the Chidamber-Kemerer metrics associated with each class she worked on during the previous seven days and trigger an email if the values of these metrics exceed developer-specified values. This enables the system to monitor the complexity of the classes that the developer works on and to send an email if they exceed the specified value.

Student usage creates an opportunity for specialized analyses and alerts. For example, analyses can help students see whether "last minute hacking" leads to more testing failures, less testing in general, and lower productivity. Alerts can help students monitor their usage and inform them when their effort falls below a certain level of consistency (such as at least one hour of effort at least 4 days a week).

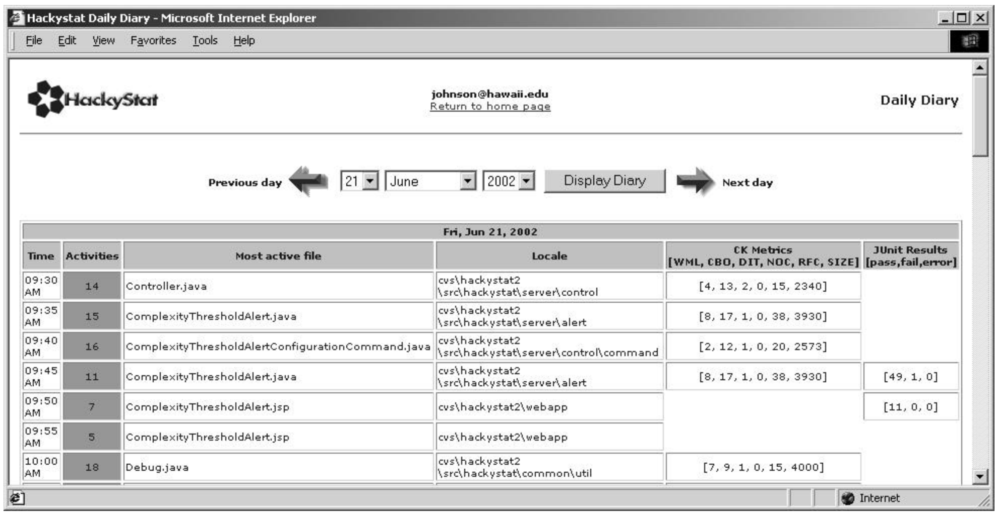
Alerts provide a kind of "just-in-time" approach to metrics collection and analysis. The developer can effectively "forget about" metrics collection and analysis during her daily work, but the metrics will still be gathered and available to her when she has a need for them. Furthermore, the alert mechanism can make her aware of impending problems without her having to regularly "poll" her dataset looking for them.

4. Results

The Hackystat project began in early 2001, and the first operational release of the server and a small set of sensors occurred in July 2001. The server is written in Java and contains approximately 200 classes, 1000 methods, and 15,000 non-comment lines of code. Client-side, tool-specific code is much smaller: the JBuilder sensor code is approximately 200 lines of Java, and the Emacs sensor code is approximately 400 lines of Lisp. Hackystat is available without charge under an open source license and is available for download at http://csdl.ics.hawaii.edu/Tools/Hackystat. In addition, we maintain a public server running the latest release of Hackystat at http://hackystat.ics.hawaii.edu/.

Hackystat is currently being used by approximately 40 students in undergraduate and graduate software engineering classes at the University of Hawaii, as well as by one industry site. One user has development data for over 250 days spanning 15 months of usage. We will be gathering adoption data regarding the ongoing use of Hackystat throughout 2003.

Our research confirms the quote that begins this paper: we did not discover a means to automatically collect PSP-style effort, size, and defect data. On the other hand, our research shows that automatic collection of Hackystat-style



Fri, Jun 21, 2002

Time	Activities	Most active file	Locale	CK Metrics [WML, CBO, DIT, NOC, RFC, SIZE]	JUnit Results [pass,fail,error]
09:30 AM	14	Controller.java	cvs\hackystat2 \src\hackystat\server\control	[4, 13, 2, 0, 15, 2340]	
09:35 AM	15	ComplexityThresholdAlert.java	cvs\hackystat2 \src\hackystat\server\alert	[8, 17, 1, 0, 38, 3930]	
09:40 AM	16	ComplexityThresholdAlertConfigurationCommand.java	cvs\hackystat2 \src\hackystat\server\control\command	[2, 12, 1, 0, 20, 2573]	
09:45 AM	11	ComplexityThresholdAlert.java	cvs\hackystat2 \src\hackystat\server\alert	[8, 17, 1, 0, 38, 3930]	[49, 1, 0]
09:50 AM	7	ComplexityThresholdAlert.jsp	cvs\hackystat2\webapp		[11, 0, 0]
09:55 AM	5	ComplexityThresholdAlert.jsp	cvs\hackystat2\webapp		
10:00 AM	18	Debug.java	cvs\hackystat2 \src\hackystat\common\util	[7, 9, 1, 0, 15, 4000]	

Figure 3. The Daily Diary: Developer metrics at five minute intervals.

effort, size, and defect data is indeed possible. It is instructive to compare and contrast the two approaches to these three metrics and its implications.

Effort. In the PSP, effort data (whether recorded by hand or using a tool) is always associated with a "project" and a "phase" of development. So, a developer might record that from 10:00am to 11:00am, she was working on Project "Timer1.2" in the phase "design". Although in theory this seems simple enough, in practice it incurs significant overhead to define unique "projects" for every development activity, determine the "phase" to be assigned, and record individual entries each time the developer switches to a different task or project. In addition, the PSP requires that you record "idle time", so every phone call or colleague's appearance at your door generates an additional recording activity.

In Hackystat, effort data is associated with active modification of a file, and has a fixed grain size of five minute increments. If the developer is not actively changing a file, then they are "idle". Instead of a "project", Hackystat has the concept of a "locale", which generally corresponds to a subdirectory (or package) hierarchy. There is currently no attempt to represent development "phase" in Hackystat.

While the PSP effort representation has the potential to be more accurate than Hackystat's, the reality is that the overhead and context switching required to conform to PSP effort collection makes it exceedingly costly to the developer. Hackystat effort data, on the other hand, is effectively "free". Another difference is in the application of effort data to planning and estimation. In the PSP, one plans using "projects" which are associated with various sizes and effort levels. In Hackystat, one can plan using "locales", which are also associated with various sizes and effort levels. However, one can also plan using simple "work week" data, which involves examining size and effort over a representative period of weeks.

Size. In the PSP, the developer invokes a source code analysis tool to collect size data at the end of project (and perhaps at the beginning, if the project is an incremental extension of an existing system). Size data consists of counts of classes, methods, and non-comment lines of code.

In Hackystat, similar size data is collected, but this data needs to be incrementally collected since there are no projects, much less defined start or end dates. This poses a problem, since the source code files parsed by the source code analysis tools are frequently syntactically incorrect while they are under active development. Hackystat solves this for Java by parsing the .class file associated with the most recent compilation of the source file. This enables Hackystat to provide size information such as the actual number of new methods added during a given day.

Defects. In the PSP, the developer must record every defect, including compilation defects, as well as the time it took to remove them, any other defects injected as a result of this defect, the phase in which the defect was injected into the product as well as the phase in which it was removed.

In Hackystat, pre-release defect data is automatically collected by attaching a sensor to a unit testing mechanism such as JUnit, and post-release defect data is automatically collected by attaching a sensor to a bug reporting system such as Bugzilla.

PSP defect collection supports a number of analyses not possible with Hackystat defect collection, such as the relationship between the cost of removal of a defect and the interval in phases between its injection and removal. On the other hand, PSP defect collection creates substantial developer overhead, and is quite sensitive to "collection fatigue". For example, if developers stop recording defects as conscientiously over time, then potentially incorrect and misleading analyses (such as a trend toward decreased defect density) can result. Hackystat defect collection is not susceptible to these problems, and does support activities such as complexity measurement validation (the development of models that predict post-release defect rates from pre-release complexity measures).

While Hackystat reduces barriers to adoption due to developer overhead, it creates a new adoption issue of its own: the specter of "Big Brother". As Figure 3 illustrates, Hackystat servers provide a fairly detailed log of developer activities, which may cause privacy concerns, particularly among professional developers who might worry about access to the data by management. We have taken several steps to address privacy. First, data access requires a password that should be known only to the developer who owns the data. Second, we maintain a public Hackystat server that allows developers to keep their data "off site" and thus unavailable to management. Third, a developer might alternatively decide to download the Hackystat server and run it locally so that all data is kept under their immediate control. We may investigate further measures, such as PGP encryption of data, if privacy issues are revealed to be a major barrier to adoption.

5. Conclusions and Future Directions

Our first conclusion is the need for further research on the issue of PSP adoption. While there now exists ample case study evidence that the PSP can provide software engineering benefits in a classroom setting, our own experience and other anecdotal evidence suggests that most developers abandon PSP practices after its use is no longer mandated.

Our second conclusion is that a significant barrier to adoption of metrics by individual developers occurs when there is the need to regularly "context switch" between product development and process recording. This indicates that second generation approaches that simply automate PSP-style effort, size, and defect collection might not be widely adopted.

Our third conclusion is that third generation approaches such as the Hackystat system present a promising means to eliminate the need for context switching by developers by automatic collection of metric data. However, the approach changes the nature of the data that is collected, and raises new adoption issues related to privacy. We hope that other researchers will download and evaluate Hackystat to explore these issues or be inspired to develop their own third (or fourth!) generation approach to metrics collection and analysis for individuals.

Hackystat is under active development, and we are currently developing sensors for Eclipse, Forte, and CVS. We are also beginning empirical studies regarding the construct validity of measures such as "Most Active File." Finally, we will be assessing the long-term adoption of Hackystat by following changes in usage patterns by students as they move on to other classes or professional work.

6. Acknowledgments

We gratefully acknowledge the hardworking and disciplined students from our current and prior software engineering courses. Support for this research was provided in part by grants CCR98-04010 and CCR02-34568 from the National Science Foundation.

References

[1] J. Borstler, D. Carrington, G. Hislop, S. Lisack, K. Olson, and L. Williams. Teaching PSP: Challenges and lessons learned. *IEEE Software*, 19(5), September 2002.

[2] W. Hayes and J. W. Over. The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Pittsburgh, PA., 1997.

[3] J. Henry. Personal Software Process studio. http://www-cs.etsu.edu/softeng/psp/, 1997.

[4] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, January 1995.

[5] P. M. Johnson and A. M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6), November 1998.

[6] P. M. Johnson, C. A. Moore, J. A. Dane, and R. S. Brewer. Empirically guided software effort guesstimation. *IEEE Software*, 17(6), December 2000.

[7] S. Khajenoori and I. Hirmanpour. An experiential report on the implications of the Personal Software Process for software quality improvement. In *Proceedings of the Fifth International Conference on Software Quality*, pages 303–312, October 1995.

[8] C. A. Moore. Lessons learned from teaching reflective software engineering using the Leap toolkit. In *Proceedings of the 2000 International Conference on Software Engineering, Workshop on Software Engineering Education*, Limerick, Ireland, May 2000.

[9] M. Ramsey. Experiences teaching the Personal Software Process in academia and industry. In *Proceedings of the 1996 SEPG Conference*, 1996.

[10] B. Shostak. Adapting the Personal Software Process to industry. *Software Process Newsletter #5*, Winter 1996.

[11] D. Tuma. PSP dashboard. http://processdash.sourceforge.net/, 2000.

[12] E. F. Weller. Lessons learned from three years of inspection data. *IEEE Software*, pages 38–45, September 1993.

[13] L. A. Williams. *The Collaborative Software Process*. PhD thesis, University of Utah, 2000.