1. Introduction

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" [5].

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.

Six years ago, 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 [3, 2]. 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.

2. Three generations of metrics for individuals

Looking back, we can divide our research on metrics for individuals into three 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 email indicated that none of them continued using the manual PSP after finishing the semester. We attributed this to the overhead involved in col-

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

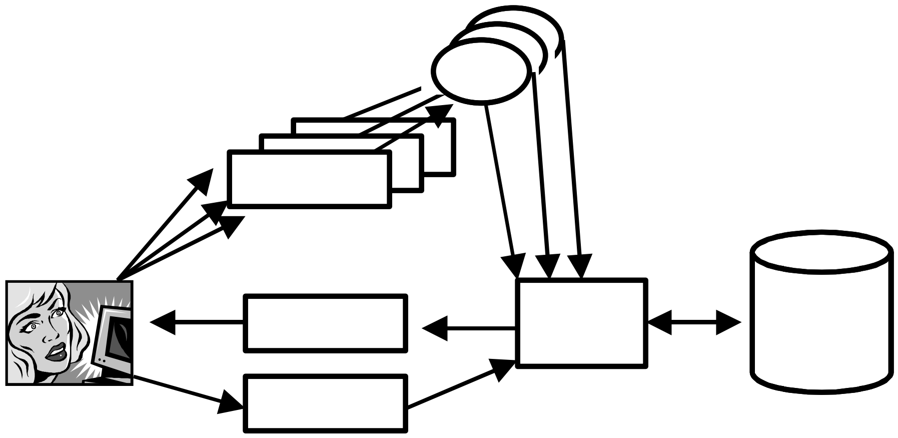


Figure 1. The basic Hackystat architecture and information flow

lection 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 [1] and PSP Dashboard [4]. 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.

3. An overview of Hackystat

Figure 1 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).

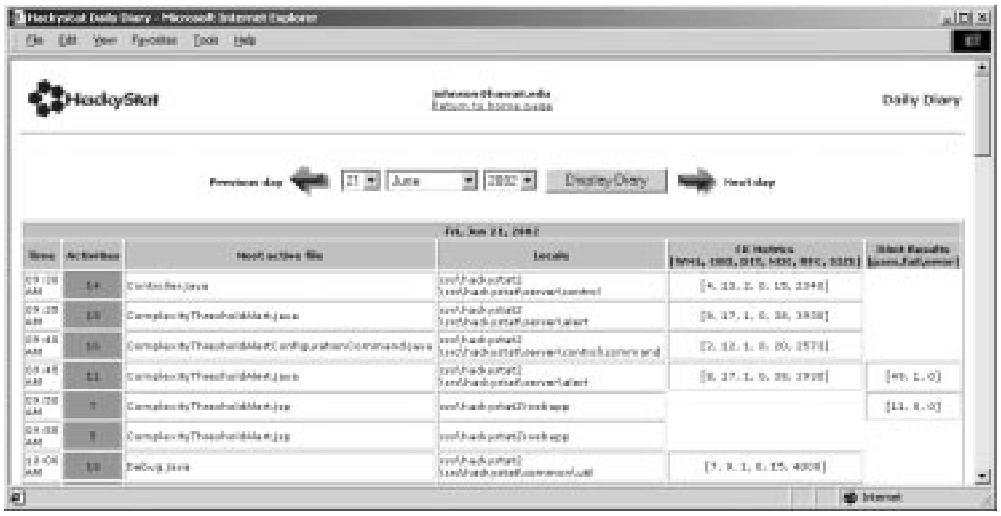
The developer begins using Hackystat by installing one or more sensors, and registering with a Hackystat server. During registration, the server sets up an account for the developer and sends her an email containing a randomly generated 12 character key that serves as her account password. This password 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 2. 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



Hackystat Daily Diary - Microsoft Internet Explorer

HackyStat [illegible]@hawaii.edu Return to home page Daily Diary

Previous day 21 June 2002 Display Diary Next day

Fri, Jun 21, 2002					
Time	Activities	Most active file	Locale	CK Metrics [WMC, CBO, DIT, NOC, RFC, [illegible]]	JUnit Results [pass,fail,error]
09:20 AM	14	Controller.java	src\hackystat2 \src\hackystat\server\control	[4, 13, 2, 0, 15, 2346]	
09:25 AM	13	ComplexityThresholdAlert.java	src\hackystat2 \src\hackystat\server\alert	[8, 17, 1, 0, 38, 1938]	
09:40 AM	16	ComplexityThresholdAlertConfigurationCommand.java	src\hackystat2 \src\hackystat\server\control\command	[2, 12, 1, 0, 20, 2571]	
09:45 AM	11	ComplexityThresholdAlert.java	src\hackystat2 \src\hackystat\server\alert	[8, 17, 1, 0, 38, 1938]	[49, 1, 0]
09:50 AM	7	ComplexityThresholdAlert.jsp	src\hackystat2\webapp		[11, 0, 0]
09:55 AM	8	ComplexityThresholdAlert.jsp	src\hackystat2\webapp		
10:00 AM	10	Debug.java	src\hackystat2 \src\hackystat\common\util	[7, 9, 1, 0, 15, 4008]	

Figure 2. The Daily Diary: Developer metrics at five minute intervals.

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 to date

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/.

References

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

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

[3] 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.

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

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