Searching under the Streetlight for Useful Software Analytics

Philip M. Johnson, University of Hawaii at Manoa

// *Developers and researchers must weigh the trade-off between easily obtained analytics and richer analytics with privacy and overhead concerns.* //

THE STREETLIGHT EFFECT is a common form of observational bias, named in honor of the following joke:

> *A drunk has lost his keys and is looking for them under a streetlight. A police officer comes over and asks what he's doing. "I'm looking for my keys," he says. "I lost them over there." The policeman looks puzzled. "Then why are you looking for them all the way over here?" "Because the lighting here is so much better."*

For more than 15 years, researchers at the Collaborative Software Development Laboratory (CSDL) at the University of Hawaii at Manoa have looked for analytics that help developers understand and improve development processes and products. Through this research, we've come to believe that the "searching under the streetlight" metaphor is useful for understanding both our research and that of others in this area.

In this context, searching under the streetlight involves collecting and analyzing metrics that are easily obtained with little social, political, or developmental impact. Unfortunately, the easier an analytic is to collect and the less controversial it is to use, the more limited its usefulness and generality. For example, collecting the data in a configuration management repository is easy, and the repository's public nature means that developers generally don't object to analysis of this data. However, the resulting analytics are constrained by the very narrow slice of development activity captured. Conversely, the original version of the Personal Software Process (PSP) can yield rich, high-impact analytics. However, it incurs significant overhead cost for developers, and the analytics have social and political implications.

Here, I provide a perspective on the CSDL's research in this area to support two claims. First, this trade-off appears to be an essential design characteristic. Second, future research is unlikely to yield a technological silver bullet that provides rich analytics without social and political implications.

It's Better to Light a Candle: The PSP

CSDL research on analytics began in 1996, when it started using and evaluating the PSP as described in Watts Humphrey's book *A Discipline for Software Engineering*.[1] This book was innovative in three main ways. First, it showed how to adapt organizational software process analytics for individual developers. Second, it showed how these analytics could drive improvement. Finally, it presented the practices in an incremental fashion amenable to academic and professional adoption.

This book's version of the PSP uses simple spreadsheets, manual data collection, and manual analysis. Collecting and managing this data takes substantial effort. In one version of the PSP, developers must fill out 12 forms, including

- a project plan summary,
- a time-recording log,
- a defect-recording log (see Figure 1),
- a process improvement proposal,
- a size estimation template,
- a time estimation template,
- a design checklist, and
- a code checklist.

0740-7459/13/$31.00 © 2013 IEEE

Name: Jill Fonson **Program:** Analyze.java

Date	No.	Type	Inject	Remove	Fix time	Fix defect no.	Description
9/2	1	50	Code	Com	1	1	Forgot import
9/3	2	20	Code	Com	1	2	Forgot ;
9/3	3	80	Code	Com	1	3	Void in constructor

FIGURE 1. A sample defect-recording log. In the Personal Software Process (PSP), even compiler (syntax) errors are recorded. Developers typically find this aspect of the PSP to be onerous.

These forms typically yield more than 500 distinct values that developers must manually calculate. Interestingly, Humphrey actively embraced the manual nature of the PSP: "It would be nice to have a tool to automatically gather the PSP data. Because judgement is involved in most personal process data, no such tool exists or is likely in the near future."[1] More fundamentally, Humphrey viewed his predefined PSP processes as a bootstrapping method. In the book, he exhorts developers to modify the forms and procedures he presents to address specific circumstances and needs.

In conjunction with our metaphor, we view this original version of the PSP as "lighting a candle" rather than looking under a streetlight because the approach promotes custom, situation-specific analytics. The manual nature of the PSP makes its analytics fragile, in the same way a candle flame is easily extinguished. On the other hand, the manual nature also makes the PSP's analytics flexible. Just as a candle enables its holder to navigate in the darkness, the PSP enables and encourages its users to search for the analytics best suited to their needs. Consider a developer who suspects that the number of interruptions he or she experiences each morning directly impacts productivity. The PSP provides explicit encouragement to explore this analytic; the techniques with which to make a sound, evidence-based conclusion; and a relatively low-cost means of doing so using just a simple spreadsheet to collect and analyze the data.

Unfortunately, after using and teaching the predefined PSP processes for two years, we suspected that the manual nature created the potential for significant data quality problems. We conducted an empirical study that checked more than 30,000 data values generated by classroom use of the PSP.[2] The manual nature of the PSP sometimes led to incorrect process conclusions despite a low overall error rate (less than 5 percent). To address this problem, we developed the Leap (*l*ightweight, *e*mpirical, *a*ntimeasurement dysfunction, and *p*ortable software process measurement) toolkit. However, as we see in retrospect, we unwittingly compromised one of the PSP's best features.

The Leap Toolkit: From Candle to Campfire

The Leap toolkit attempts to address the data quality problems we encountered with the PSP by automating and normalizing data analysis.[3] Although the developer still manually enters most data, the toolkit automates subsequent PSP analyses and in some cases provides analyses (such as various forms of regression) that the PSP doesn't provide. The approach is lightweight because it doesn't prescribe the sequence of development activities (unlike the PSP). It attempts to avoid measurement dysfunction by enabling developers to control their data files. It maintains data about only the individual developer's activities and doesn't reference developers' names in the data files. Leap data is also portable. It creates a repository of personal process data that developers can keep with them as they move from project to project and organization to organization. Figure 2 illustrates a Leap component that supports time estimation on the basis of personal historical data and selection of a regression analysis.

In our metaphor, the Leap toolkit replaces the PSP candle with a campfire. Introducing higher-level tool support metaphorically increases the light by improving data quality and decreasing the manual analysis required. On the other hand, unlike a candle, whose light can be moved around according to the holder's interests, a campfire is stationary; participants must come to it. By introducing automation, the Leap toolkit makes certain analytics easy to collect but others increasingly difficult. Consider our hypothetical developer who suspects that interruptions are affecting productivity. He or she would now be expected to design and implement a new Leap toolkit component rather than a simple spreadsheet form.

After several years of using the Leap toolkit, we came to agree with Humphrey that the PSP approach could never be fully automated and would inevitably require significant manual data entry. We also came to agree with the agile community that such development overhead frequently doesn't provide enough return on investment. This is particularly true when each project significantly differs from the previous one, so as to render historical data inappropriate for comparison.

Our next project, however, departed from the conventional wisdom of both

camps. Unlike the PSP and TSP (Team Software Process) community, we abandoned any pretense of supporting PSP analyses. Unlike the agile community, we would continue to embrace extensive measurement and analysis. The research question was simple: What kinds of useful software analytics could we obtain if both collection and analysis were "free"? Answering it became the mission of a decade-long research project called Hackystat.

Hackystat: The Harsh Glare of Operating-Room Lights

As users of both the PSP and the Leap toolkit, we were personally aware of the development overhead such data collection creates, notwithstanding downstream benefits in the form of better planning and reduced defects. Conventional wisdom says to define high-level goals first and then figure out the data collection and analysis necessary to achieve them.[4] The Hackystat project went in the opposite direction.[5] We first focused on developing ways to collect software process and product data with little to no overhead for developers. We then determined what high-level software engineering goals could be supported by analyses on this data. Hackystat implements a service-oriented architecture in which sensors attached to development tools gather process and product data and send it to a server, which other services can query to build higher-level analyses.

Hackystat includes four important design features. The first is both client- and server-side data collection. Modern software development typically includes individual developers' activities on their local workstation as well as server- or cloud-based activities. From the start, we developed instrumentation for client-side tools such as editors, build tools, and test tools, as well as server-side tools such as configuration management repositories, build servers, and so on.

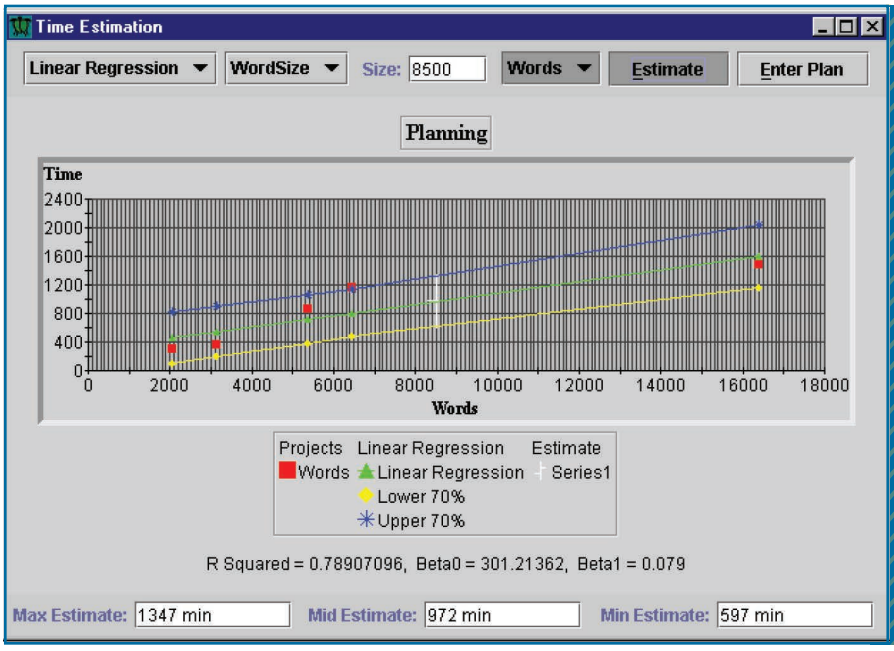


FIGURE 2. The time estimation component in the Leap (*l*ightweight, *e*mpirical, *a*ntimeasurement dysfunction, and *p*ortable software process measurement) toolkit. Unlike the PSP, no Leap analytics are paper-based.

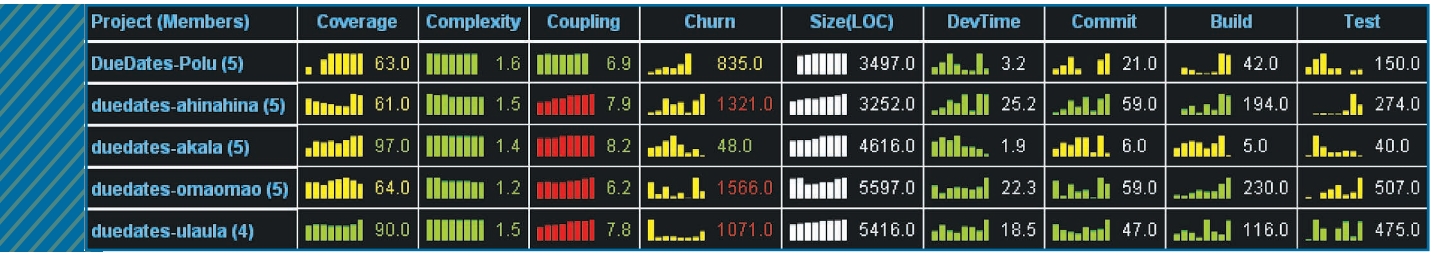
The second feature is unobtrusive data collection. For developers, one of the most frustrating aspects of manual data collection is the loop of doing some work and then interrupting it to record what they worked on. An important requirement for Hackystat was to make data collection as unobtrusive as possible. Users shouldn't notice that data is being collected, and the system shouldn't make assumptions about network availability. For example, Hackystat client-side instrumentation locally caches any data collected while a developer works offline. It then sends the data to the Hackystat data repository when the developer reconnects.

The third feature is fine-grained data collection. By instrumenting client-side tools, we can collect data on a minute-by-minute or even second-by-second basis. For example, Hackystat supports a measurement called *buffer transition*—collecting a data instance each time the developer changes the active buffer from one file to another. Hackystat can also track a developer as he or she edits a method, constructs a test case for that method, and invokes the test, yielding insight into real-world test-driven development.

The fourth feature is both personal and group-based development. Besides collecting their personal development data, developers can define projects and shared artifacts to represent group work. Hackystat can track the interplay among developers when, for example, they edit the same file.

Hackystat has led to a variety of technical innovations, including

- the development of a toolkit for defining and visualizing software project telemetry,[6]
- support for high-performance-computing software development,[7]
- a method for prioritizing which software development artifacts to inspect,[8]



Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (5)	63.0	1.6	6.9	835.0	3497.0	3.2	21.0	42.0	150.0
duedates-ahinahina (5)	61.0	1.5	7.9	1321.0	3252.0	25.2	59.0	194.0	274.0
duedates-akala (5)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0
duedates-omaomao (5)	64.0	1.2	6.2	1566.0	5597.0	22.3	59.0	230.0	507.0
duedates-ulaula (4)	90.0	1.5	7.8	1071.0	5416.0	18.5	47.0	116.0	475.0

FIGURE 3. A Software ICU (intensive care unit) display based on Hackystat. The Software ICU assesses a project's health both alone and in relation to other projects.

- an operational definition for test-driven development,[9]
- an approach to software process discovery,[10] and
- the Software ICU (intensive care unit), which assesses a project's health both alone and in relation to other projects (see Figure 3).[11]

These strengths have been noticed. Sixth Sense Analytics (a start-up company later acquired by Borland) incorporated Hackystat technology into a commercial offering in 2006. Also, University of Bolzano researchers developed a similar technology called PROM (PRO metrics).[12]

However, our research on Hackystat uncovered three significant social or political problems with this approach. First, although we viewed the unobtrusive nature of data collection as a feature, some developers considered it a bug. They didn't want to install instrumentation that would collect data regarding their activities without telling them about it.

Second, client-side, fine-grained data collection can create discord in a development group. One user called the Software ICU "hacky-stalk," complaining about the transparency it provided regarding each member's working style.

Third, the client-side, fine-grained data that provides the most compelling analytics about development is also the largest obstacle to industrial adoption of Hackystat technologies. Developers repeatedly informed us that they weren't comfortable with management access to such data, despite management promises to use it appropriately. (Robert Austin has provided more details on this problem.[13])

A closer look at the Software ICU helps explain these problems. As the left side of Figure 3 shows, the Software ICU collects and displays software artifacts' structural metrics such as coverage, complexity, coupling, and churn. It colors the most recently observed values and trends red, yellow, or green to indicate health. Another structural metric is size, which the Software ICU displays for informational purposes but colors white (because size trends don't indicate health). The Software ICU displays these values for a portfolio of projects, allowing project data comparison. In general, collection, analysis, and public presentation of the values to the left of the size data aren't controversial.

Things get interesting on the Software ICU interface's right side, which presents four health indicators based on aggregations of individual developer behavior:

- *DevTime* estimates how much time each developer spends in his or her IDE (integrated development environment) working on each file associated with the project.
- *Commit* measures how often each developer commits to the repository and how many lines of code he or she commits each time.
- *Build* measures how many times each developer builds the system and whether each build is successful.
- *Test* measures how often each developer invokes the test suite on the system and whether the tests ran successfully.

For a more detailed perspective, users can click on any sparkline. For example, clicking on the DevTime sparkline generates a visualization showing each developer's DevTime trend.

CSDL research suggests that such a representation of individual developer behavior makes some developers uncomfortable; however, it's necessary to provide certain kinds of insight. For example, a principle of agile software development is "build early and often." The Software ICU can measure the extent to which developers adhere to this.

The Hackystat-based Zorro system provides an even more sophisticated application of developer behavior data. It can automatically determine the extent to which developers use test-first design methods. Such an analysis requires a fine-grained, second-by-second analysis of developer behavior (see Figure 4). Once again, some developers were uncomfortable with this fine-grained data collection.

Returning to our metaphor, Hackystat provides the equivalent of

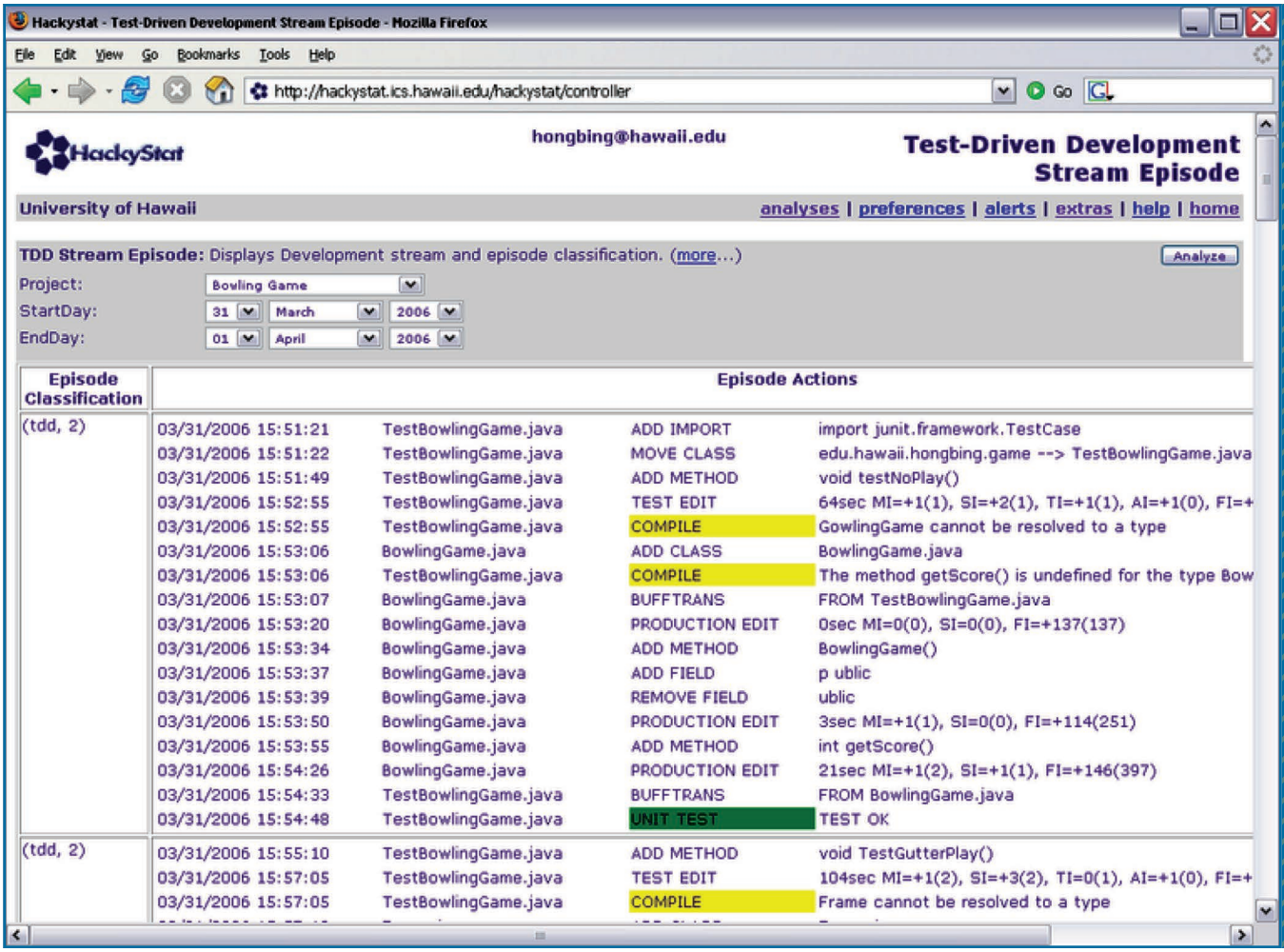


FIGURE 4. The Hackystat-based Zorro system can automatically determine the extent to which developers use test-first design methods. Some developers were uncomfortable with this fine-grained data collection.

high-intensity operating-room lights. It offers the potential for abundant illumination and deep insight, but these benefits are often out of reach without procedures some might view as invasive. Furthermore, the Hackystat philosophy of automated collection would make it exceedingly difficult for our hypothetical developer who suspects that interruptions are impacting productivity. To fit the philosophy, he or she would need to design and implement some combination of hardware and software to automatically and unobtrusively detect a workflow interruption (for example, a coworker knocking on the developer's office door). The technology would then need to send data about the interruption's start and end times to a Hackystat server for further analysis.

The State of the Practice: Back under the Streetlight

Over the past few years, services for software product analytics have become popular, with offerings from DevCreek, Ohloh, Atlassian, CAST, Parasoft, McCabe, Coverity, Sonar, and others. These services' analytics are typically built from one or more of three basic sources: a configuration management system, a build system, and a defect-tracking system. Figure 5 shows a display from Sonar for the SpringSource project, which is representative of this type of service.

These systems have two significant strengths. First, data collection is entirely automated, and the data is already available. The service simply applies analytic techniques (coverage, complexity, security, and so on) to the data and displays results in a friendly user interface. Because the data is automatically gathered from a repository,

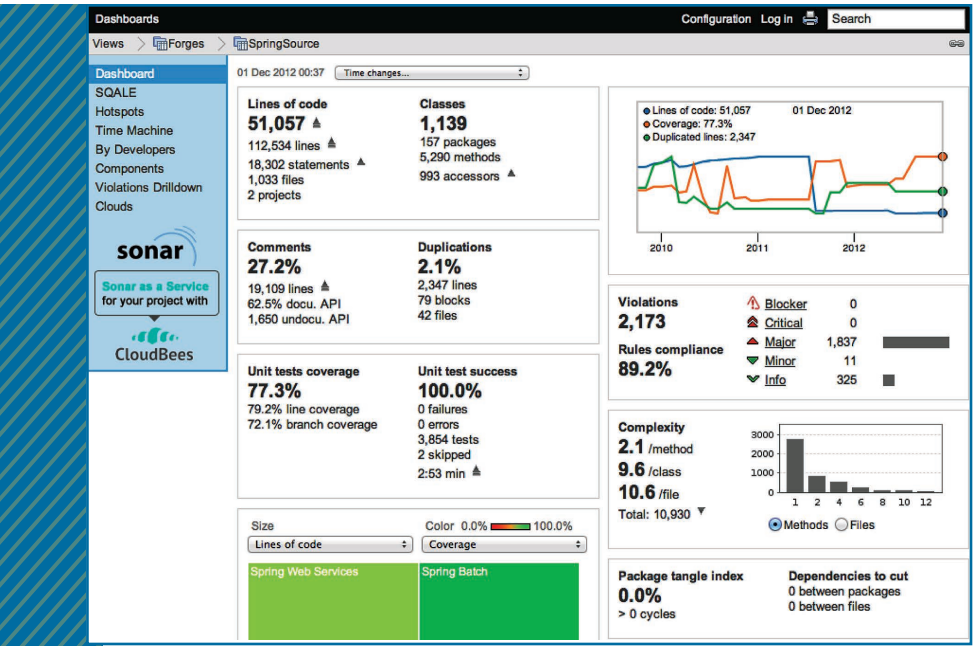


FIGURE 5. The Sonar dashboard display, showing a collection of product metrics. Sonar is representative of the current crop of popular services for software product analytics.

overhead for developers and managers is low. Second, the data is generally uncontroversial; it focuses on product characteristics, not the developer behaviors that produced them.

These systems correspond best to the metaphor of searching under the streetlight because that's where the light is. They're optimized for easy installation and integration, but the result is rather limited illumination of software processes and products. For example, our hypothetical developer who suspects that interruptions are affecting productivity is entirely out of luck. The approach simply doesn't support such behavioral, client-side data collection and analysis. In addition, none of these systems can offer insight into the use of developer practices such as test-driven development.

A Matter of Trade-offs

Figure 6 summarizes our experiences; it illustrates the trade-offs in designing analytics for software processes and products, which involve three dimensions:

- The degree of automation and the level of overhead developers and management incur to obtain the analytics.
- The barrier to adoption incurred by the technique or technology, which could be social or political. At its worst, this barrier could lead to measurement dysfunction, entirely undermining the analytic.
- The technique or technology's level of generality (represented by the size of the circles in Figure 6). That is, how broad or narrow is the range of analytics that can be developed while adhering to the technique or technology's essential characteristics?

As you can see, the PSP, Hackystat, and modern product analytic technologies such as Sonar occupy three separate quadrants in Figure 6. Agile measurements (such as velocity, burndown, and burn-up) fit in the fourth quadrant. In the parentheses are analytics that would be difficult to implement with techniques or technologies in the other quadrants.

After many years of exploring different approaches to analytics, we conclude that the field isn't converging on a single best approach, nor are the latest approaches intrinsically better than earlier ones. Rather, the community has been exploring the space of trade-offs among expressiveness, simplicity, and social acceptability.

Consideration of the various approaches suggests three fruitful directions for future research and practice. First, current approaches such as Sonar aren't necessarily advancements over older approaches such as the PSP, nor is the PSP obsolete. They simply make different trade-offs. Developers who suspect that interruptions are impacting productivity won't find Sonar data helpful. That said, certain aspects of the original PSP (such as recording syntax errors) are probably no longer useful in the age of IDEs such as Eclipse.

Second, a hybrid approach that mixes the best of automated collection and analysis with carefully chosen, high-impact manual data entry by developers could substantially increase the analytics' impact, with acceptable overhead for developers.

Finally, modern approaches to privacy could assuage some developers' fears regarding behavioral data collection and analysis. Consider a cloud-based, independent, privacy-oriented analytics repository in which developers could maintain complete control over data and choose whether to provide management access. Just as companies establish privacy mechanisms to encourage whistleblowers to come

forward, companies could decide that the benefits of insightful software analytics warrant giving developers increased control over their own data.

Acknowledgments

These findings result from the hard work of Collaborative Software Development Laboratory (CSDL) researchers, including Joy Agustin, Robert Brewer, Joe Dane, Anne Disney, Jennifer Geis, Austin Ito, Aaron Kagawa, Honging Kou, Christoph Lofi, Carleton Moore, Mike Paulding, Dan Port, Julie Sakuda, Pavel Senin, James Wang, Cedric Zhang, and Shaoxuan Zhang. The US National Science Foundation has been a primary sponsor of this research through grants 9403475, 9804010, and 0234568. All CSDL software was developed using open source licensing. For more details on the research and access to the software, visit http://csdl.ics.hawaii.edu.

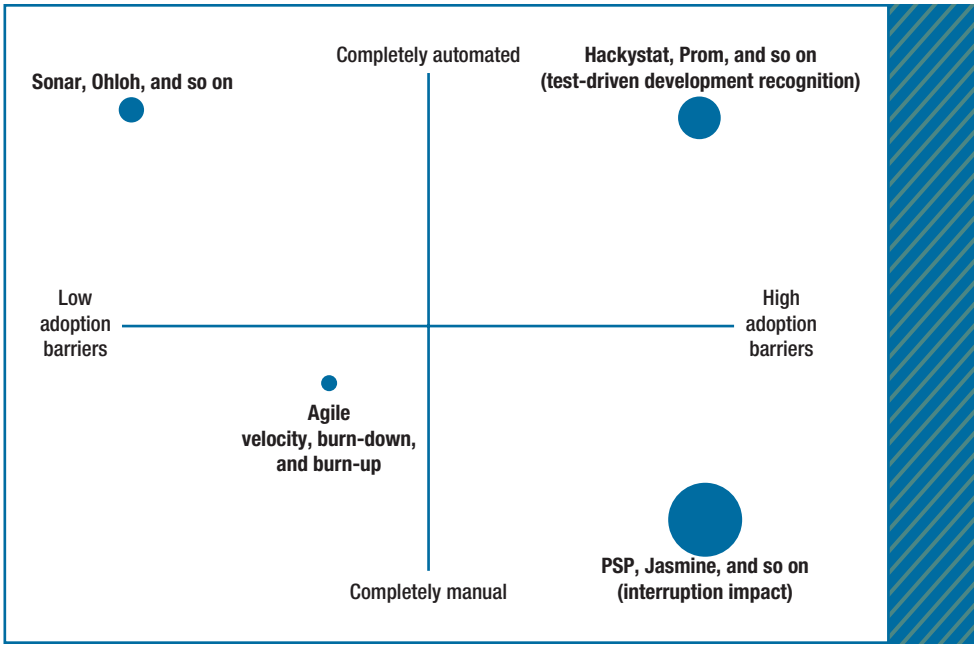


FIGURE 6. A classification for software analytics approaches, including automation, adoption barriers, and the breadth of possible analytics the approach supports (indicated by the circles' size). In the parentheses are analytics that would be difficult to implement with techniques or technologies in the other quadrants.

References

1. W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, 1995.
2. P.M. Johnson and A.M. Disney, "The Personal Software Process: A Cautionary Case Study," *IEEE Software*, vol. 15, no. 6, 1998, pp. 85–88.
3. P.M. Johnson, "Leap: A 'Personal Information Environment' for Software Engineers," *Proc. 21st Int'l Conf. Software Eng.* (ICSE 99), IEEE CS, 1999, pp. 654–657.
4. V. Basili, G. Caldiera, and H.D. Rombach, "The Goal Question Metric Approach," *Encyclopedia of Software Eng.*, J.J. Marciniak, ed., John Wiley & Sons, 1994.
5. P.M. Johnson et al., "Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS, 2003, pp. 641–646.
6. P.M. Johnson et al., "Improving Software Development Management through Software Project Telemetry," *IEEE Software*, vol. 22, no. 4, 2005, pp. 76–85.
7. P.M. Johnson and M.G. Paulding, "Understanding HPCS Development through Automated Process and Product Measurement with Hackystat," *Proc. 2nd Workshop Productivity and Performance in High-End Computing* (P-PHEC 05), IEEE CS, 2005; https://csdl-techreports.googlecode.com/svn/trunk/techreports/2004/04-22/04-22.pdf.
8. A. Kagawa, *Priority Ranked Inspection: Supporting Effective Inspection in Resource-Limited Organizations*, tech. report 2005-08-02, Information and Computer Science Dept., Univ. Hawaii at Manoa, 2005.
9. H. Kou, P.M. Johnson, and H. Erdogmus, "Operational Definition and Automated Inference of Test-Driven Development with Zorro," *J. Automated Software Eng.*, vol. 17, no. 1, 2009, pp. 57–85.
10. P. Senin, *Software Trajectory Analysis: An Empirically Based Method for Automated Software Process Discovery*, tech. report 09-09, Collaborative Software Development Lab, Univ. Hawaii at Manoa, 2009; https://csdl-techreports.googlecode.com/svn/trunk/techreports/2009/09-09/09-09.pdf.
11. P.M. Johnson and S. Zhang, "We Need More Coverage, Stat! Experience with the Software ICU," *Proc. 3rd Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 09), IEEE CS, 2009, pp. 168–178.
12. I.D. Coman, A. Sillitti, and G. Succi, "A Case-Study on Using an Automated In-Process Software Engineering Measurement and Analysis System in an Industrial Environment," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS, 2009, pp. 89–99.
13. R.D. Austin, *Measuring and Managing Performance in Organizations*, Dorset House, 1996.

ABOUT THE AUTHOR

PHILIP M. JOHNSON is a professor and the associate chair of the Department of Information and Computer Sciences and the director of the Collaborative Software Development Laboratory at the University of Hawaii at Manoa. His research interests include software metrics, software engineering, smart grids, gamification, and human-computer interaction. Johnson received a PhD in computer science from the University of Massachusetts. Contact him at johnson@hawaii.edu.