Project Hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis

Philip Johnson

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

November 5, 2001

Contents

1 Executive Summary

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.

This report describes Hackystat, a technology initiative and research project that explores the strengths and weaknesses of a *developer-centric*, *in-process*, and *non-disruptive* approach to empirical software project data collection and analysis. Hackystat makes available to developers a set of custom sensors that they voluntarily attach to their development tools. Once installed, these sensors automatically monitor characteristics of the developer's process and products and send data to a centralized web service. The web service maintains a repository of process and product data for each developer, performs analyses on the repository, and automatically sends the developer an email when new, unexpected, and/or potentially interesting analysis results become available.

Hackystat is developer-centric because all data is collected directly from developer activities, and all analyses provided back to that same developer. It is in-process because data is collected regularly throughout project development, and analysis results are returned and intended to be useful during development. It is non-disruptive because developers do not need to interact directly with the sensors during development to enable collection or analysis.

Hackystat is designed to accelerate adoption of empirically guided software project measurement by providing a new approach to to addressing the barriers of cost, quality, and utility identified above. We will evaluate the Hackystat project through the following research components:

1. *Bootstrap and ongoing technology development.* The "bootstrap" phase will create a critical mass of sensors and analysis mechanisms required for experimentation. Additional development will occur throughout the project.

2. *Verification and validation.* Verification focuses on assessing the fidelity of the sensors; in other words, does a sensor that is intended to detect "idle time" actually detect it with sufficient accuracy to support related analyses? Validation focuses on assessing 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*. We will contrast our approach with the Personal Software Process (PSP): a developer-centric, in-process, *disruptive* approach to software project data collection and analysis.

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.* By the end of the three years of the study, 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.

2 Project Description

2.1 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.

For example, Cocomo [6] and Cocomo II [8] provide models of software development that enable organizations to predict the cost in resources and schedule time given characteristics of the proposed project. An essential phase of Cocomo model development is calibration, where empirical software project data from organizations is collected and used to determine internal model parameters. The Personal Software Process (PSP) [23] is a method for improving software project planning and software quality assurance for individual programmers by collection and analysis of empirical software project data. The Team Software Process (TSP) [24] builds upon the data collected by the PSP with additional analyses to support group-based software development. The Goal-Question-Metric paradigm (GQM) [3] provides a methodology for collection and analysis of software project data with explicit traceability to organizational goals, from which an Experience Factory [2] can be built. Both the Fagan [13] and Gilb [18] Inspection methods use collection and analysis of software review data to improve the efficiency and effectiveness of future reviews. The Capability Maturity Model for Software (SW-CMM) [36] requires organizations to track project cost, schedule, and functionality to reach Maturity Level 2, and to provide quantitative measures of process and quality at Maturity Level 4.

While these approaches and others explore the potential benefits of empirically guided approaches to software development, effective collection and analysis of software project data is rare in mainstream software development. While a wide variety of factors contribute to this situation [38], studies suggest 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.

For example, in the development of the Cocomo II model, over 2000 datasets were gathered from industrial organizations for the purposes of calibrating the model, yet only 161 of them were sufficiently accurate and complete enough to be used for calibration [7]. A case study of the PSP uncovered over 1500 errors made by 19 developers during data collection and analysis [28]. One study found that the total average effort to introduce GQM-based measurement is approximately one person-year [17]. Extreme Programming (XP) eschews most traditional project and process measures as requiring too much developer effort and leading to insufficiently timely feedback [26]. A case study of industrial metrics programs revealed very low developer confidence in the accuracy of software project metrics [20]. One reason cited for the failure of the National Software Data and Information Repository was the perceived high cost of data submission and low value of benefits returned [19]. A 1990 survey of 300 major US information technology companies implementing measurement programs determined that only 60 could be viewed as "successful". Reasons for the failure of the remaining 240 programs included: the measures were viewed as irrelevant, developers thought measures might be used against them, and data collection was too time-consuming [40].

The problems associated with the cost, quality, and utility of empirical software project data creates problems for research, because it raises the expense of empirical research and the time required for study results to feed back into practice. It also create problems for industry, because many organizations do not believe they can afford the resources required to collect and analyze empirical software project data or believe the effort will be cost-effective.

This report describes Hackystat, a technology initiative and research project that explores the strengths and weaknesses of a *developer-centric*, *in-process*, and *non-disruptive* approach to empirical software project data collection and analysis. In essence, Hackystat makes available to developers a set of custom sensors that they voluntarily attach to such development tools as their editor, source code control system, unit testing

framework, and so forth. Once installed, these sensors automatically monitor characteristics of the developer's process and products and send data using SOAP and XML to a centralized web service. The web service maintains a repository of process and product data for each developer, performs analyses on the repository, and automatically sends the developer an email when new, unexpected, and/or potentially interesting analysis results become available.

Hackystat is *developer-centric* because all data is collected directly from developer activities (such as writing code, running test cases, checking in documents to a repository, executing the system, encountering defects, and so forth). Analyses are also developer-centric, in that they are oriented toward the immediate interests and needs of developers. For example, Hackystat might email the developer that test case coverage of a particular module has decreased significantly due to recent enhancements. Finally, data access is limited to the developer who generated it. Other measurement approaches often require the involvement of non-developers to collect and/or analyze the data, or result in data that is of only marginal use to developers.

Hackystat is *in-process* because data is collected regularly throughout project development. Analysis results are also in-process, in that they are provided to developers throughout project development and are meant to help guide in-process changes. In contrast, many other measurement approaches are effectively "between-process", in that they collect data after completion of a project and analyze it in order to improve development on a future project.

Hackystat is *non-disruptive* because developers do not have to stop what they are doing in order to tell a measurement tool what data should be collected about what they are currently doing. Hackystat sensors do not interrupt developers or required them to shift their focus of attention from their primary task. Analysis results from Hackystat are also non-disruptive; they arrive as email so that developers have control over when and whether to respond to them. This contrasts with other "disruptive" in-process measurement approaches such as the Personal Software Process [23].

The overall goal of the Hackystat project is to accelerate adoption of empirically guided software project measurement by providing a new approach to addressing the barriers of cost, quality, and utility identified above. Hackystat addresses cost by requiring that all empirical software project data be collected and analyzed automatically through sensors and without any developer or manager involvement. Eliminating human involvement in data collection and analysis also addresses several documented problems with quality, such as when the high overhead of manual collection and analysis leads to incomplete data [28] or to the "massaging" of data [20]. By making the system responsible for detecting interesting or anomalous conditions in the data and by notifying the developer by email when this occurs, the approach ensures that the design of data collection and analysis is tied directly to developer utility.

This project goal and approach creates a new set of research questions and risks. Clearly, automated measurement lowers the cost of data collection, but is the data collected automatically sufficiently accurate for these purposes? Certain kinds of project data can't be sensed automatically; what limitations does this place on the utility of the analyses? Will commercial tools expose the appropriate API for Hackystat sensors? Finally, will the privacy and security mechanisms suffice to prevent adoption problems due to the specter of "Big Brother"? The Hackystat project is designed to answer these and other questions by gathering metrics of the system's accuracy, utility, and adoptability in academic and industrial settings.

The basic components of Hackystat are available. Over the past six months, we have implemented the web service and an initial set of sensors and associated analysis mechanisms. The system is in daily use and the sensors and web service are available for public download [27].

Figure 1 illustrates some of the initial features of the 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

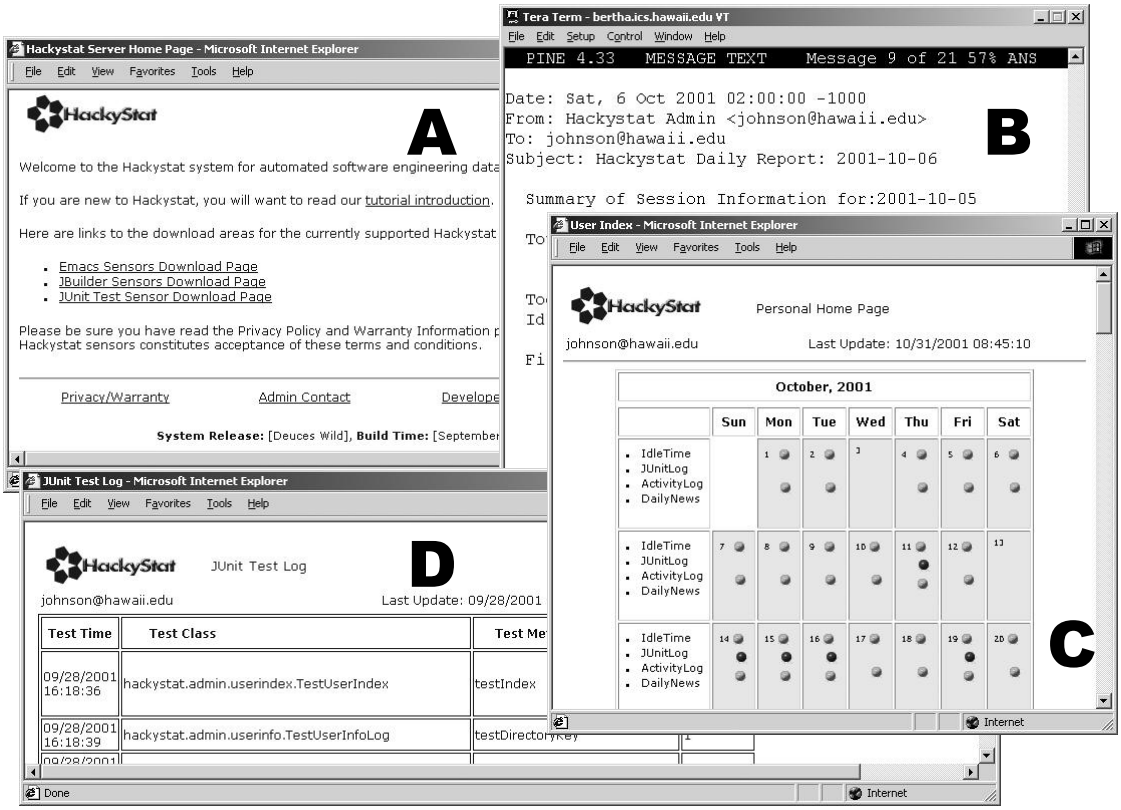

Figure 1: *Sample screenshots illustrating the current Hackystat implementation. Screen A shows the web server home page with links to download sensors. Screen B illustrates an email to a developers. Screen C shows what data has been collected on any particular day, with green button links indicating "normal" and red button links indicating "interesting" data. Screen D shows a specific log file obtained by clicking on a button.*

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.

The approach embodied in Hackystat has both evolutionary and revolutionary aspects. It is evolutionary because its design reflects our research results in empirical software engineering over the past ten years. It is revolutionary because various empirical software engineering research has concluded that totally automated metrics collection and analysis can't and/or shouldn't be expected or attempted [23, 17, 48].

We will pursue the goals of the Hackystat project through the following research components:

1. *Bootstrap and ongoing technology development.* The "bootstrap" phase is focused on creating a critical mass of sensors and analysis mechanisms to initiate verification, validation, and experimentation. Additional development and refinement of sensors and analyses will occur continuously throughout the project.
2. *Verification and validation.* Verification focuses on assessing the fidelity of the sensors; in other words, does a sensor that is intended to detect "idle time" actually detect it with sufficient accuracy to support related analyses? Validation focuses on assessing 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. This case study will explore the strengths and weaknesses of disruptive vs.

non-disruptive 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 both introductory and advanced computer science courses at the University of Hawaii. By the end of the three years of the study, we will have gathered in-process software development data from students as they progress from introductory to advanced programming. This data will be analyzed to provide insights into the development of advanced programmers, with the goal of improving educational practice.

These components will be described in detail in the research plan, Section 2.5. The intervening sections describe results from our previous NSF-funded research and how they motivate the Hackystat project; discuss influences of other research on this project; and document the current status of the project.

2.2 Results from prior NSF-sponsored research

The motivation for and design of our proposed research on developer-centric, ultra-lightweight empirical software project data collection and analysis follows directly from the findings of our last two NSF-sponsored research projects.

2.2.1 The CSRS project

In this research, we designed, implemented, and evaluated a collaborative software review system called CSRS and performed a series of experiments with the following contributions.

Design of instrumented review technology. The CSRS system illuminates the design and architectural issues involved with the creation of a review system with both fine-grained measurement support and a process modeling language able to express a wide variety of current software review techniques. For example, in addition to traditional review metrics such as the number of issues discovered during different review phases, CSRS allows one to measure the time spent on review of any single function in the source code, and the sequence of functions visited by a reviewer over the course of review. The CSRS process modeling language allows emulation of most review techniques such as Fagan inspection [13] and Active Design Reviews [35], as well as new review techniques such as FTArm that require an online environment.

Instrumentation support for experimentation. The measurement support and process flexibility provided by CSRS also provides excellent infrastructure for controlled experimentation. We used it to investigate the costs and benefits of meetings in software review by implementing two review techniques, one which involved a synchronous meeting between review team participants to discuss issues, and another in which all team participant interactions were asynchronous. Our study found that the meeting-based method was significantly more costly in terms of effort than the non-meeting-based method, though we could not detect any difference in the numbers or types of defects found.

Comparison with Maryland experimental data. To gain insight into the generality of our findings on the efficiency and effectiveness of software review meetings, we teamed with Professor Adam Porter of the University of Maryland to perform a modified form of meta-analysis we call "reconciliation" on the data resulting from the CSRS-based experiment and an independently designed and conceived experiment performed by Porter at the University of Maryland. Our reconciliation process differs from traditional meta-analysis in that we did not attempt to combine the data sets; instead, we formulated a set of common hypotheses and tested them against each data set independently. Despite differences in the types of documents reviewed, the backgrounds of the participants, and the technological infrastructure support between the two experiments, the two studies confirmed each other's results for all six common hypotheses, providing further evidence that meeting-based review may not be the most efficient and effective choice for many development contexts.

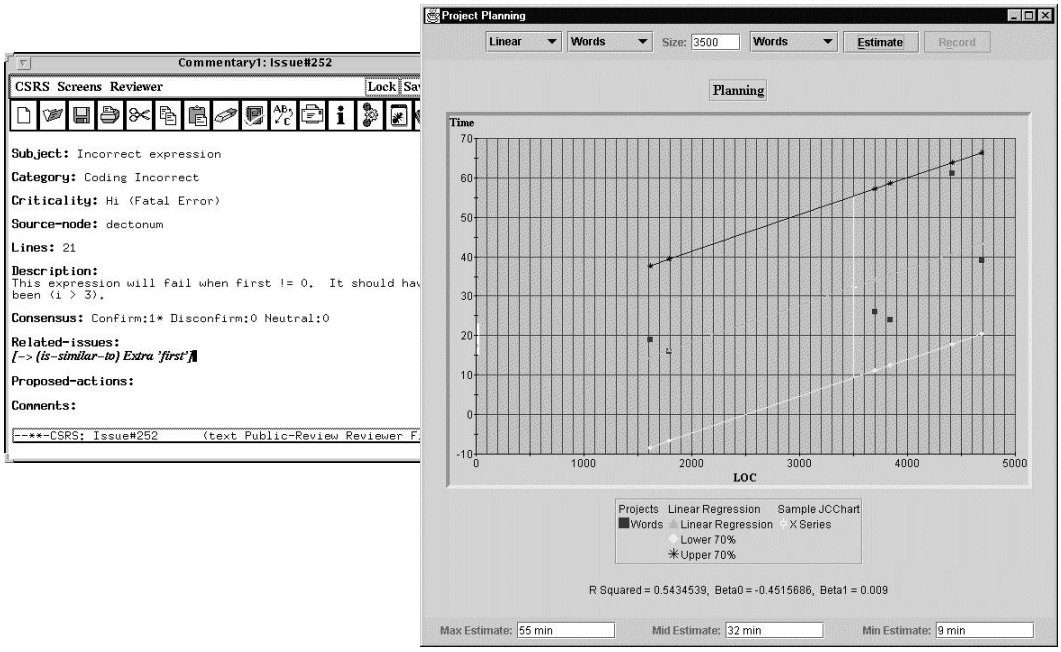

Figure 2: *Screen shots illustrating the Emacs-based CSRS interface for issue reporting (left) and the Java-based Leap interface for project size and time estimation (right).*

Usability issues in review technology and measurement infrastructure. In addition to the above positive findings, our research also uncovered usability problems with CSRS that impeded its adoption in academia and industry. The feedback we received indicates four principle usability problems. First, CSRS requires a Unix back-end server for its hypertext database engine, and some sites did not have a Unix server available. Second, the client interface to CSRS is tightly integrated with Emacs, as illustrated in Figure 2, and many potential users find Emacs difficult to use. Third, the hypertext back-end and Emacs front-end constrained review documents to those available in ASCII format; while this constraint was unimportant for source code review, it often proved problematic for design or requirements documents. Fourth, our fine-grained metrics collection and its attendant storage in a centralized repository raised concerns among developers that the data would be used inappropriately by management. After a conversation with one manager in which he spoke excitedly about the extensions he envisioned for CSRS, we realized that developer fears might well be justified.

2.2.2 The Leap project

In this research, we built upon the results of the CSRS project as well as upon a case study of data quality in the Personal Software Process (PSP). In the case study, we subjected the data collected manually by students during the standard semester-long introduction to the PSP to a retrospective analysis of its accuracy. We implemented a database system to cross-check all calculations and and expose six classes of data defects. The analysis discovered over 1500 errors made by the 10 students, errors that in several cases lead to incorrect inferences concerning process improvement.

Based upon the usability problems we found with CSRS and the data quality problems we found with the PSP, we designed, implemented, and evaluated a Java-based toolkit called Leap. Leap satisfies four principal requirements. First, it is *lightweight*: it imposes a minimum of process constraints upon the user, it should be easy to learn and integrate with other methods and tools, and require minimal investment and commitment from management. Second, it is *empirical*, providing high quality collection and analysis of quantitative data.

Third, it reduces the risk of *measurement dysfunction* [1], or the situation in which developers consciously or unconsciously skew data collection and analysis for personal, political or organizational goals. Finally, the toolkit and associated data is *portable*, allowing the user to continue to build their personal repository of software engineering data as they move within and across development organizations.

The Leap project is nearing completion, with the following contributions:

Technology support for developer-centric, in-process, disruptive software project data collection and analysis. The Leap toolkit has been implemented and evaluated both through classroom and industrial use. The toolkit consists of approximately 40,000 lines of Java code and runs on all major platforms. The tool supports the collection of time, size, defect, and design pattern data, as well as the definition of size types, defect types, and projects and their associated components. The design of the Leap toolkit eliminates broad classes of data quality problems that users experience in the manual PSP.

Case study results on project estimation. The project estimation component of Leap provides 13 different possible estimation methods, in contrast to the PSP, which implements a single estimation method called PROBE. We performed a case study using the Leap toolkit in which we were able to retrospectively assess the estimation accuracy of the 13 methods. There were two surprising results. First, the PROBE method was sixth out of the 13 with respect to data accuracy. Second, the most accurate estimation technique was a form of "guesstimation", in which the developers, aided by the empirical data gathered by the tool and the results of the various estimation techniques, simply entered their best estimate.

Usability improvements over CSRS. The Leap toolkit addressed many of the usability problems we discovered in our research on CSRS. Leap runs on Unix and Windows platforms and provides a GUI interface, as illustrated in Figure 2. Rather than a centralized server, Leap stores data files locally so that the user has total control over their own data. Leap does not provide a collaborative review mechanism and does not store software development artifacts internally, so there is no restriction to ASCII representation.

Usability problems due to "disruptive" in-process data collection. Despite the advances in automation and usability, Leap has still found limited adoption. Few students have continued to use it after the end of the semester in which use was mandated, despite their acknowledgment that the data collected and analyzed using Leap did improve their project estimation and quality assurance capabilities. We are in contact with a group of industrial developers voluntarily using Leap, but they appear to be much more highly motivated than the typical developer to collect and analysis personal software project data.

Our successes and failures with CSRS and Leap deeply influence the design of the current research project, and give us an understanding of the variety of requirements that must be simultaneously addressed in order for a measurement-based system to achieve widespread, long-term adoption by developers across diverse organizations. First, the system's interface must be very simple and require little initial training in order to derive non-trivial benefits. Second, those who contribute the data must be those who derive benefit from the data, and those benefits must occur in the near-term, not the long-term. Third, empirical software project data is highly susceptible to measurement dysfunction, and mechanisms must be available to provide developers with the confidence that the data will not be used against them in the future. Fourth, and perhaps most importantly, even when developers experience substantial benefits from daily collection and regular analysis of empirical project data under mandated conditions, most begin "forgetting" to collect and analyze their data as soon as they are left on their own. In the case of PSP, we conjectured that this was due to the high overhead of manual daily collection and analysis. However, our experiences with Leap tool support indicate to us that the fundamental adoption problem of developer overhead will not be solved through normal, evolutionary approaches to measurement automation, such as GUI-based utilities to simplify daily entry of time, size, and defect data and associated analyses for project planning or post-mortems.

2.3 Related work

This section describes relationships to work by other researchers and how it influences the proposed project.

Measurement theory and data validity in empirical software engineering. Research by Fenton, Bush, Hall, Pfleeger, Kitchenham, and others has improved understanding of effective measurement collection and analysis in empirical software engineering [9, 14, 20, 31, 32, 38]. One influence of this work on the design of Hackystat involves data comparability. For example, Hackystat requires that all sensors for active development time collect data in the same way, although they may be written in different languages for different tools.

Research by Basili, Fuggetta, Rombach, and others on data validation and the Goal-Question-Metric (GQM) paradigm stresses the need for validation procedures and explicit ties between low-level measurements and high-level organizational goals [3, 2, 5, 17, 44]. In Hackystat, validation procedures are required to ensure that the model of user behavior as represented by the sensor data is sufficiently accurate for the purposes of the analyses. In addition, users need to know why the measures are being collected in order to feel comfortable allowing the sensors to be active.

Finally, Fenton and Neil present a critique of traditional applications of software measurement which provide simplistic correlations between such measures as size and effort, as well as pre-release and post-release quality [15]. The Hackystat project is designed to provide the kinds of data appropriate for either simplistic correlations (when such correlations are actually helpful in the context of developer-centric measurement), as well as the kinds of more sophisticated causal analysis mechanisms they advocate.

Statistical process control in empirical software engineering. Another influential branch of research involves the use of statistical process control techniques to assess and improve the stability and predictability of software development procedures [16, 47]. The normal application of statistical process control techniques is to drive the software development process toward a predictable state, where future values of measures can be reliably estimated and historical variations are within "natural" limits. In Hackystat, statistical process control techniques (such as control charts) will be applied to support developers in understanding whether or not the measures under collection are "in-control" or not, and thus whether it would be appropriate to use them in estimation. However, Hackystat does not require processes to be "in-control" in order for the system to be useful.

Agile software development methods. Newly emerging software development methods such as Extreme Programming, Scrum, Crystal methods, adaptive software development, and others are critical of traditional approaches to project measurement as requiring too much developer overhead with little immediate benefit [22, 26]. Hackystat is designed to appeal to this community by providing extremely low cost measurements oriented around the needs of developers, and by providing feedback within a development iteration. For example, Hackystat already provides a sensor for JUnit that collects data on test case invocation and its results.

The Personal Software Process (PSP) and Team Software Process (TSP). Research on the PSP and TSP strongly influence the design of this research. First, the PSP and TSP demonstrate the importance and potential utility of collecting size, time, and defect data. With sufficiently accurate and complete historical data concerning these three primary measurements, the PSP and TSP show how developers can generate a variety of analyses that can lead to substantial improvements in both their project planning capabilities and in the ultimate quality of their software.

Despite these promising results, case studies of the PSP suggest that the level of in-process interruption required by the methods creates fundamental adoptability problems [28, 45]. Hackystat will explore whether automated data collection and analysis can be used to provide most of the utility of the PSP in a manner suited to a broad spectrum to developers and organizational contexts.

Measurement toolkits. Other toolkits have been created for empirical software project data collection and analysis, though we know of no system satisfying our developer-centric, in-process, and non-disruptive requirements.

A variety of toolkits collect PSP-inspired data from developers through in-process disruption. Examples of these systems include PAMPA [42], the PSP Process Dashboard [39], the PSP Studio [21], and the Excel spreadsheet distributed with the Team Software Process [24].

The Balboa system supports in-process collection of software engineering data [11]. These streams of event data are analyzed to infer and/or validate process models via grammar-based rules. Thus, Balboa can aid developers in answering questions like, "Was every minor release preceded by a formal code inspection?", though developers must manually decide when to ask these questions and what questions to ask.

Research on the Amadeus system explored a set of architectural principles and abstract interfaces for designing metric-driven analysis and feedback systems [41]. Amadeus was intended to include active agents, a custom scripting language, and custom client-server components, but little experience with a working implementation has been reported.

The Ginger2 system provides sensors for detecting keystrokes and mouse gestures, eye traces, three dimensional movement, skin resistance levels, and video recording of the developer [43]. Unlike Hackystat, whose sensors are designed for deployment "in the field", Ginger is a laboratory-based environment.

Finally, MetricCenter [4] is a commercial measurement toolkit provided by Distributive Software. Like Hackystat, MetricCenter provides sensors that send data to a central repository for analysis. Unlike Hackystat, the sensors and analyses performed by MetricCenter are management-centric, not developer-centric.

Software project metrics repositories. The Hackystat server functions as a metrics repository, and benefits from prior research in this area. The International Software Benchmarking Standards Group maintains a public repository with data on over 1,200 projects from over 20 countries, including information on project context such as the business area, product characteristics such as the user base, development characteristics such as the programming language, size characteristics such as function points, as well as others [33]. Hackystat leverages insights from ISBSG such as the importance of anonymity in gaining industrial cooperation, though Hackystat data is developer-centric, not management-centric.

A less successful metrics repository initiative is the National Software Data and Information Repository (NSDIR). While a variety of factors appear to have contributed to the failure of this initiative, one issue was "that program managers, under pressure to complete projects on time and on budget, did not perceive participation in NSDIR would yield benefits equaling the effort.[19]." In the Hackystat project, in contrast, the overhead of participation is designed to be extremely low, so that a positive cost-benefit analysis is easier to obtain for individuals.

2.4 Current project status

Development of the Hackystat system began in May, 2001, and the first functional public release was made in July, 2001. This release included source and binaries for the Hackystat server, a set of sensors for detecting time and activity data in JBuilder and Emacs, and server-side analysis tools for summarizing time and activity data. In addition, a publically accessible Hackystat server was deployed on the Internet and has been available and regularly updated with new releases since then. Since the initial release, additional sensors and analysis tools have been released at monthly intervals. The active development team consists of five members, three from the University of Hawaii and one member from each of two commercial software development companies.

The Hackystat server code is implemented in Java and contains approximately 14 packages, 62 classes, and 4,200 non-comment source lines of code. The code leverages over a dozen open source components including Tomcat for web services, Cocoon for XML/XSLT processing, JUnit for testing, and SOAP for sensor-server communication over the HTTP protocol.

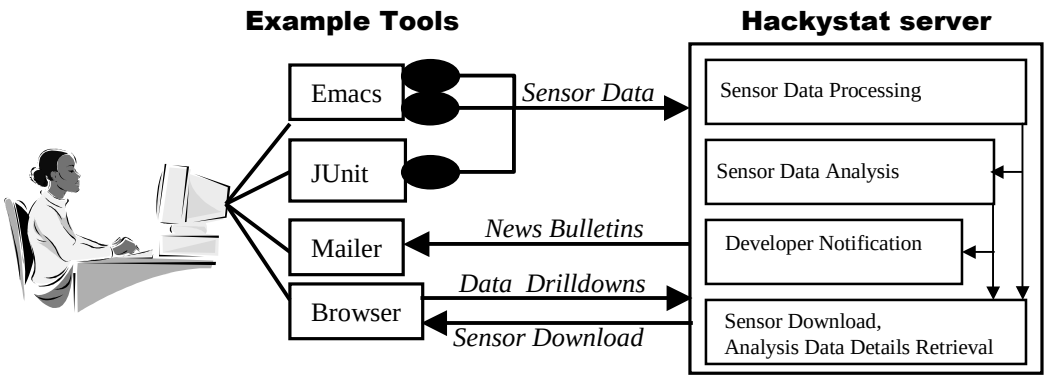

Figure 3: *Basic communication pathways in Hackystat.*

Hackystat sensor code is much smaller and simpler than that required for the server. For example, the JBuilder activity and session sensors consist of a total of approximately 200 non-comment source lines of Java, with over half of that code consisting of ”empty” method declarations required to fulfill plug-in API requirements. The Emacs activity and session sensors are written in Lisp and are similar in size.

Currently implemented sensors include: time and activity data for the Emacs and JBuilder development environments, and defect data for JUnit. An initial implementation of a size data sensor for Java and C++ using the LOCC grammar-based size counting tool will be available in November, 2001. Currently implemented analysis tools include active time data, idle time threshold analysis, JUnit summary data, and data integrity checking for all data types.

Hackystat does not use a relational database back-end. Instead, data for each user for each sensor data type is stored in a “log” file in XML format. Currently, some developer log files store only a day’s worth of data (in the case of potentially high volume data such as activity and testing data), while other logs store a month’s worth of data (in the case of low volume data such as session data). Note that different sensors can contribute data to the same log file; for example, activity data for a given developer on a given day is stored in the same log file, even if some of that data was sent from an Emacs sensor and other data was sent from a JBuilder sensor. Each log file type includes data integrity code which checks incoming data from sensors. If bad data is detected, it is not added to the sensor log. Instead, it is added to a special “BadData” log and the user and system administrator gets information concerning this in their next daily email. Bad data should not be sent by correctly implemented sensors under normal circumstances, of course. We use the JATO COTS component to support automated translation between the XML log file and an in-memory Java object representation, which simplifies the implementation of both log files and analysis functions. While we are pleased with both the flexibility and performance of our XML-based approach so far, we could easily migrate to a relational database approach in future if that becomes necessary.

Initial performance evaluation results are encouraging. There are four important costs that impact upon performance: the cost of collecting data in the sensor, the cost of sending sensor data to the server, the cost of performing analyses on the data, and the cost of storing data.

If any of these costs become excessive, they affect the usability or scalability of the system. First, the cost of collecting data by the sensor is so far negligible for all sensors we have implemented. Second, sensor data is typically aggregated and sent to the server at the end of tool sessions. Currently, this means that Hackystat sends data from a developer less than a dozen times a day, and each notification normally requires less than two seconds for the http interaction. Third, analysis computations are performed once a day during “off-hours” such as 2 am, and current analyses require only a few seconds in total to perform for each user. We believe that none of these first three costs prevent the current system architecture from scaling to at least several hundred users per server with many additional sensors, sensor data types, and analyses.

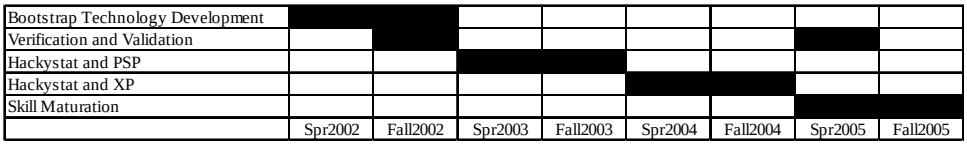

Figure 4: *Project Timeline.*

The fourth and final cost, data storage, is the only one which currently appears to pose significant scalability threats. Currently, a typical Hackystat user accumulates data in XML log files at the rate of 500 KB per month. As the set of sensors and thus data gathering potential increases, we believe that data accumulation could easily double to 1MB/user/month, which could create performance problems as the number of users rises to hundreds or more. We know of one relatively simple potential solution: use built-in Java utilities to read and write the XML data as compressed zip files. Tests indicate that Hackystat logs are compressed by an average of 90% using zip encoding.

Figure 3 illustrates the basic communication pathways and processing in Hackystat. First, the developer downloads one or more sensors from the server and installs them into the development environment. Next, all sensors require some initial configuration; at a minimum, the developer must specify the URL of the Hackystat host server to which data should be sent and an email address identifying them as the user. At that point, sensors begin sending data to the server, and the server notifies the developer via email when potentially "interesting" analyses occur. The developer then has the option of retrieving additional pages from the server containing analysis details.

Our progress so far indicates to us that Hackystat represents a feasible approach to developer-centric, in-process, non-disruptive software project data collection and analysis, and that our research group has the technical skills required to build and enhance such a system. However, our work so far has only begun laying the groundwork for answering important questions about this approach. For example, we know we can collect data automatically, but can we collect the right data in the right way? We know we can send the developer an email with analysis results, but can we send the developer an email at the right time with the right analysis results? The next section presents our approach to addressing these issues.

2.5 Research plan

We now present the principal components of the research plan: bootstrap and ongoing technology development; verification and validation; a comparative study of data collection and analysis in Hackystat and the PSP; a case study of automated data collection and analysis for Extreme Programming; and a longitudinal study of software development skill maturation. A time-line for the project is shown in Figure 4.

2.5.1 Bootstrap and ongoing technology development

Technology development will be an ongoing activity throughout the project. "Bootstrap" technology development refers to the subset of technology development required before the verification and validation component can begin.

First, we will provide implementations of the ten sensor data types listed in Figure 5. We have already identified many subtle issues in the definition and implementation of these sensor data types [30]. Most importantly, we recognize that every sensor data type is partial: the system cannot collect the total project effort of a developer, or all of the defects that occur, or all of the times a certain file is edited or a system is compiled. A key validation task is to ensure that useful developer feedback can be provided even in the face of partial data about the product and process.

Sensor Data Type	Description
ToolTime	The time the developer is "busy" using a development environment tool. ToolTime data appears to be accurate only to within 15 minutes for a given session.
IdleTime	The time during which a development environment tool is running but the user is not actively engaged with it such that activities are being recorded.
Activity	An event that occurs while using a development environment tool. Activities are used both to detect idle time and to detect events such as file modification, compilation errors, and so forth.
File Name	The name, including the directory path, of a file manipulated by a development environment tool.
Module	An aggregation of files. Modules can be identified through explicit representations such as "projects", or implicitly through common directory structures and co-occurrence within recorded activity sequences.
Total Size	Refers to a collection of sensor data types that calculate different representations of the total size of a software product. Our initial total size measure will leverage our prior research results on structural size measurement and the resulting grammar-based tool called LOCC [12].
Differential Size	Measures the difference in size between two successive versions of a system. LOCC implements one approach to differential size measurement that has been successfully applied to incremental project planning.
C/K Complexity	Measures the six Chidamber-Kemerer object-oriented metrics: Weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, response for a class, and lack of module cohesion [10]. Initial language targets are Java and C++.
Defect/Syntax	The unsuccessful compilation of a file due to a syntax error. The sensors will not attempt to infer what specific syntax error occurred or how many were in the file.
Defect/Runtime	The unsuccessful termination of a program due to a run-time error. Hackystat can determine only that a run-time error occurred and the "top-level" reason (such as "Null pointer exception").
Defect/Unit test	Invocation of a specific unit test and its result (pass/fail).
CM/Tag	An aggregation of files that have been tagged as belonging to a release.

Figure 5: *Summary of sensor data types.*

Second, we will implement a set of analyses over these sensor data types. To satisfy our developer-centric, in-process, and non-disruptive criteria, analyses will be implemented as a server-side computation over the sensor log files collected for each developer, and which will result in an automatically generated, developer-specific "Hackystat News Bulletin" email containing analysis results. Analyses are only run over developer logs when data from that developer was received during the previous day and an email is sent only when analysis results from the previous day indicate the data is of interest.

The current implementation supports the News Bulletin email, but the current analyses are simplistic, consisting of simple summaries of data collected and in some cases a comparison of these values to past weekly or monthly averages. For development purposes, we currently view receipt of any data during the previous day as sufficiently "interesting" for the email. However, developers quickly get used to such "boilerplate" summary emails and stop reading them carefully. To be effective, Hackystat must identify important changes in the sensor data and bring those changes to the attention of the affected developer.

One approach to identifying certain kinds of important changes is the use of statistical process control techniques. For example, appropriate control charts can enable the system under certain circumstances to detect when recent data becomes "significantly different" from past data. As a simple example, suppose the

number of weekly unit test failures has jumped from 8 to 17 to 21 during the past three weeks. Control chart data can help determine whether this is naturally occurring variation of no concern, or an important change that should be brought to the attention of the developer.

Figure 6 provides examples of the kinds of analyses that we have identified. Each row in the table lists the set of sensor data types involved in the analysis, the derived measure, and some of the "interesting conditions" that could result and the developer response. In the figure, "UCL" refers to the "Upper Control Limit" of a control chart and "LCL" refers to "Lower Control Limit". Referring to a measure as "newly in-control" indicates that a sufficient number of values of that measure have been collected to assess control, and that various characteristics of the measure indicate that it has been caused by an in-control process. Details on the procedures used to assess in-control measures are available [16, 46]. So far, we have identified 15 derived measures from combinations of data types, which have the ability to detect 24 potentially anomalous and thus "interesting" development conditions [30].

Sensor Data Type(s) *Derived measure*	**Condition(s) and implication(s)**
ToolTime *Weekly Average*	• If newly in-control, then email developer that ToolTime appears to be in-control and thus may be useful for prediction/planning purposes. • If in-control and recent value greater than UCL, then email developer that recent ToolTime values may not be sustainable. • If in-control and recent value less than LCL, then email developer that progress may be impacted by low ToolTime; may indicate deterioration of work environment.
Differential Size, Module, ToolTime *CodingRate*	• If newly in-control, then email developer that rate of coding appears to be in-control and thus may be useful for prediction/planning purposes. • If in-control and recent value greater than UCL, then email developer that coding rate is significantly higher than normal, could indicate need to review code more carefully; need to re-estimate, etc. • If in-control and recent value lower than LCL, then email developer that coding rate is significantly lower than normal, indicating potential problems with application domain; need for additional support; need to re-estimate; etc.
Unit Test, Module *Total Weekly Tests*	• If newly in-control, then email developer that total number of test cases being run per week appears to be in-control. • If in-control and recent value greater than UCL, then email developer that number of tests run this week is higher than normal, potentially indicating problems with quality of code. • If in-control and recent value lower than LCL, then email developer that number of tests run this week is lower than normal, indicating potential need to run more tests.

Figure 6: *Sample measures derived from one or more sensor data types.*

A key design goal of Hackystat is scalability with respect to the numbers of sensors and analyses. Even with the current small number of implemented sensors and analyses, the amount of data collected and potentially available to developer inspection is overwhelming. Because sensors must run without developer interaction, and because all Hackystat analyses must be designed to inform the developer only when something "interesting" has occurred with respect to that measure, growth in sensors and analyses should result in richer and more useful feedback to developers. Assessing whether that design goal is true in practice is the subject of the next research component, verification and validation.

2.5.2 Verification and validation of Hackystat data collection and analysis

Once initial technology development is complete, we will begin the first of two phases of verification and validation.

Verification activities focus on assessing the fidelity of the sensors. In other words, to what extent does the automatically collected data accurately reflect what the developers are actually doing? For example, to what extent does the proposed implementation of the ToolTime and IdleTime measures model actual time spent in the tool and actual time spent idle? To study this question, we plan to follow methods employed in a study of time usage at AT&T Bell Labs, in which both time diaries and direct observation were used to obtain insight into the daily activities of developers [37]. In our case, however, we will take the further step of correlating the data gathered manually with the data automatically gathered by the sensors. Time diaries are a standard part of the PSP curriculum, so we can easily collect (student programmer) data of this form. The direct observation component will be accomplished as part of a graduate course in either software engineering or human-computer interaction, and will involve either students or professional developers in local high tech companies.

Validation activities follow verification, and focus on assessing the utility of the sensors and analysis mechanisms. We will assess utility by investigating two research hypotheses: (1) Do developers perceive Hackystat analysis feedback to be useful? (2) Do developers actually make changes based upon the feedback they receive? To test these hypotheses, we will build a validation data gathering mechanism into Hackystat to query selected developers by email regarding the utility of the analyses they receive. Say, for example, that Hackystat sends a developer an email on a certain day indicating that their recent test failure rate has exceeded the level expected under normal development conditions. One week later, Hackystat might randomly select that specific user and analysis result for follow-up validation, and send an email requesting that the developer follow the enclosed URL to a survey page at the Hackystat server web site. This page would contain a copy of the specific analysis result reported to the developer the previous week, and ask the user to indicate whether they found the analysis result to provide useful insight into development, and if so, whether or not they actually made any changes during the past week as a result of the message. To validate the utility of Hackystat, at least half of the randomly selected analysis results should be reported as useful, and at least one quarter should have precipitated change in developer activities.

As we expect significant maturation in the capabilities of the sensors and servers over the course of the project, we will perform verification and validation activities twice, once near the beginning of the project and once near the end. The initial verification and validation activities will involve a small number of participants and the results will be used internally to prioritize system development tasks. By the end of the project, we plan to have several hundred active users of the system from which to select participants for verification and validation activities. At this time, we will augment the validation survey with a request for developers to voluntarily provide demographic information regarding technical background, organizational context, and application domain so that we can better understand the nature of our user community and the conditions under which Hackystat is found to be useful.

2.5.3 A comparative study of data collection and analysis in Hackystat and the PSP

There are two primary goals for data collection and analysis in the Personal Software Process: (1) To support the PROBE method for size and time estimation of software projects; and (2) To support defect prevention activities through defect data collection and classification. The PSP method is pessimistic about the prospects for fully automated data collection and analysis: “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. [23].” PSP thus defines itself as an intrinsically “disruptive” approach to in-process data collection.

The goal of this study is not to confirm or refute the disruptiveness of PSP, because our experience with the PSP confirms the author's view that the kind of data collected by the PSP is not amenable to total automation. Instead, our study will investigate the costs and benefits of disruptive vs. non-disruptive data collection and analysis through the following four research hypotheses: (1) Does disruptive (PSP) effort data collection lead to significantly more accurate effort estimation than non-disruptive (Hackystat) effort data collection? (2) Do developers perceive disruptive (PSP) defect data collection and analyses as significantly more useful than non-disruptive (Hackystat) defect data collection and analysis? (3) Is the PROBE size and effort estimation method significantly more accurate than alternative, simpler approaches to estimation? (4) Does disruptive, in-process data collection and analysis (PSP) have significantly lower levels of adoption than non-disruptive, in-process data collection and analysis (Hackystat)?

To test these hypotheses, we will employ a case study experimental design using approximately 20 student subjects in a graduate-level software engineering class. The course will include the standard components of the one-semester introduction to the Personal Software Process. The course will employ two forms of tool support: (1) the standard disruptive in-process PSP tool implemented as an Excel spreadsheet into which users manually enter time, size, and defect data, and distributed by the Software Engineering Institute [34], and (2) the non-disruptive in-process Hackystat tool implemented as sensors into the student's development environment. For the first half of the course involving five programming projects, students will not have access to their Hackystat data, and will be required to use the PSP tool and PROBE estimation method. During the second half of the course, they will continue to be required to gather and input data into the PSP tool, but will also have access to their Hackystat data and the additional dozen estimation methods used in the Leap case study [29].

After the semester is over, we will perform a retrospective analysis of the PSP and Hackystat data sets, comparing their relative accuracy with respect to effort estimation. The data will also allow us to perform a partial replication of the Leap study regarding the performance of PROBE relative to other, simpler estimation methods. We will gather qualitative data to investigate the perceived utility of disruptive vs. non-disruptive defect data collection and analysis. Finally, we will perform a survey of the students four months after the semester to see which (if any) of the students continue to use the PSP tool, and which (if any) of the students continue to use Hackystat, and their reasons for their usage.

2.5.4 A case study of automated data collection and analysis for Extreme Programming

Extreme Programming (XP) is perhaps the best known of the so-called "Agile Programming Methods" that have recently attracted attention in the software engineering community. XP embraces automated technology when it adds clear value to developers; indeed, XP could not exist without automated unit testing such as those provided by the xUnit framework family [25]. Yet, XP method proponents eschew traditional disruptive, in-process data collection and analysis. The primary goal of this case study is to determine if a developer-centric, non-disruptive, in-process approach to software engineering data collection and analysis is compatible with the practices of the XP development community. If so, our secondary goal is to determine if our data collection and analysis techniques can shed new light on the strengths and weaknesses of XP.

To carry out this case study, we will solicit participation from industrial XP development groups through XP-related web sites, news groups, and through presentations at XP conferences and workshops. Through this process, we hope to obtain commitment from at least five XP development groups to participate in a trial use of the Hackystat toolkit over a two month period.

Our research hypotheses for this case study are similar to those for general validation: (1) Do XP developers perceive Hackystat analysis feedback to be useful? (2) Do XP developers actually make changes based upon the feedback they receive? In addition to the data collected through the built-in Hackystat validation data gathering mechanism, we will also conduct on-site interviews with XP developers after two months of use. Although access to the actual development data collected by the Hackystat server might yield additional

useful information, we will not depend upon such access for this study and organizations will be free to install a private Hackystat server to maintain confidentiality if they prefer.

2.5.5 A longitudinal case study of software development skill maturation

Over the three years of the project, we will incrementally deploy Hackystat into introductory and advanced programming classes at the University of Hawaii. By the end of the project, we will have gathered data from the same students at multiple points along their path through the computer science degree program. We expect to obtain data covering four separate semesters from at least 100 undergraduate and graduate students by the end of the project.

The availability of this longitudinal data enables us to carry out a case study that can provide new understanding of how people change over time in the way they develop software and the kind of software they develop. This increased understanding can help point the way to more effective and/or efficient methods for educating software developers.

This case study will be retrospective, and will involve analysis of the Hackystat project data collected from students over the course of the study. We will analyze the database to find the set of students satisfying the criteria of having collected Hackystat data covering at least four separate semesters, then contact them to obtain permission to use their data in this case study. If they accept, we will then solicit demographic data from them to support the analysis. This data will include the grades they received in classes for which Hackystat data is available, along with their subjective evaluations of their skill at various aspects of software development during each semester, including time management, design, implementation, testing, and documentation.

Our research hypotheses are as follows: (1) Do the numbers and proportions of defect data (such as runtime errors and compilation interval) change as a person gains development experience? (2) Do the numbers and proportions of defect data correlate with academic achievement? (3) Do the values of complexity metrics (such as Chidamber-Kemerer's) grow as a person gains development experience? (4) How does the size of software projects change, and does the proportion of size change correlate with academic achievement?

2.6 Summary of anticipated contributions

We expect this research on non-disruptive, developer-centric, in-process software project data collection and improvement to make the following contributions to the academic and industrial software engineering communities.

1. We will provide an open-source, extensible, community-supported environment for sensor-based data collection and web service-based analysis of software project data. If Hackystat is successful, we anticipate that tool vendors will begin developing and maintaining their own sensors.
2. We will provide and maintain a publically available Hackystat web server. This will substantially lower the cost of evaluation of the services for initial users and provide developers an external "safe haven" for their process and product data.
3. We will replicate previous case study results on the relative accuracy of various approaches to individual project estimation. This will provide additional data on the merits of simpler approaches to estimation than PROBE.
4. We will contribute empirical results regarding the relative utility and effectiveness of the PSP and Hackystat approaches to data collection and analysis. This will provide insight into the strengths and weaknesses of disruptive and non-disruptive approaches to developer-centric software project data.
5. We will contribute case study results regarding the utility and adoptability of the Hackystat approach to data collection and analysis within the XP development method and community.

6. We will contribute insights into the use and applicability of statistical process control techniques to developer-centric in-process project data.
7. We will obtain empirical in-process data as students progress from beginning to advanced programming capabilities, and explore how this data can be used to improve education.

References Cited

[1] Robert D. Austin. *Measuring and Managing Performance in Organizations*. Dorset House Publishing, 1996.

[2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *Encyclopedia of Software Engineering*, chapter Experience Factory. John Wiley and Sons, 1994.

[3] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, SE-14(6):758–773, 1988.

[4] Peter Baxter. The MetricCenter toolkit. Distributive Software, Fredricksburg, Virginia, 2001.

[5] Andreas Birk, Rini van Solingen, and Janne Jarvinen. Business impact, benefit, and cost of applying gqm in industry: An in-depth, long-term investigation at schlumberger RPS. In *Proceedings of the Fifth International Symposium on Software Metrics*, Bethesda, Maryland, November 1998.

[6] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[7] Barry Boehm. COCOMO II data validation. In *Presentation at the 2001 Meeting of the International Software Engineering Research Network*, Strathclyde, Scotland, August 2001.

[8] Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford Clark, Ellis Horowitz, Ray Madachy, Donald Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[9] Martin Bush and Norman E. Fenton. Software measurement: A conceptual framework. *Journal of Systems and Software*, 12:223–231, 1990.

[10] Shyam Chidamber and Chris Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), June 1994.

[11] Jonathon E. Cook and Alexander L. Wolf. Balboa: A framework for event-based process data analysis. In *Proceedings of the Fifth International Conference on the Software Process*, pages 99–110, Lisle, Illinois, June 1998.

[12] Joseph A. Dane. Modular program size counting. M.S. thesis, University of Hawaii, December 1999.

[13] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[14] Norman Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, 1997.

[15] Norman E. Fenton and Martin Neil. Software metrics: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[16] William A. Florac, Robert E. Park, and Anita D. Carleton. Practical software measurement: Measuring for process management and improvement. Technical Report CMU/SEI-97-HB-003, Software Engineering Institute, April 1997.

[17] Alfonso Fuggetta, Luigi Lavazza, Sandro Morasca, Stefano Cinti, Giandomenico Oldano, and Elena Orazi. Applying GQM in an industrial software factory. *Software Engineering and Methodology*, 7(4):411–448, 1998.

[18] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[19] Gregory Goth. NSDIR: A legacy beyond failure. *IEEE Computer*, 18(5), September 2001.

[20] Tracy Hall and Norman Fenton. Implementing effective software metrics programs. *IEEE Software*, 14(2):55–65, March / April 1997.

[21] Joel Henry. Personal software process studio. http://www-cs.etsu.edu/psp/, 1997.

[22] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *IEEE Computer*, September 2001.

[23] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, New York, 1995.

[24] Watts S. Humphrey. *Introduction to the Team Software Process*. Addison-Wesley, New York, 2000.

[25] Ron Jeffries. xUnit automated unit testing software. http://www.xprogramming.com/software/, 2001.

[26] Ron Jeffries, Ann Anderson, and Chet Hendrikson. *Extreme Programming Installed*. Addison-Wesley, 2001.

[27] Philip M. Johnson. Hackystat system. http://csdl.ics.hawaii.edu/Research/Hackystat/.

[28] 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.

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

[30] Philip M. Johnson, Carleton A. Moore, and Jitender Miglani. Hackystat design notes. Technical Report ICS-TR-01-04, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, June 2001.

[31] Barbara Kitchenham, Robert Hughes, and Stephen Linkman. Modeling software measurement data. *IEEE Transactions on Software Engineering*, 27(9):788–804, September 2001.

[32] Barbara Kitchenham, Sheri Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, December 1995.

[33] Chris Lokan, Terry Wright, Peter Hill, and Michael Stringer. Organizational benchmarking using the ISBSG data repository. *IEEE Software*, 18(5), September 2001.

[34] James Over. PSP/TSPi support tool. http://www.awl.com/cseng/0-201-47719-x/, 2000.

[35] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. *Proceedings of Eighth International Conference on Software Engineering*, pages 132–136, August 1985.

[36] Mark Paulk, Charles Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, New York, 1995.

[37] Dewayne Perry, Nancy Staudenmayer, and Lawrence Votta. *Process-Centered Environments*, chapter Understanding and Improving Time Usage in Software Development. John Wiley and Sons, 1995.

[38] Shari L. Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham. Status report on software measurement. *IEEE Software*, 14(2):33–43, 1997.

[39] Ken Raisor and David Tuma. Process dashboard for PSP. http://processdash.sourceforge.net/, 2001.

[40] H. Rubin. The Rubin Review, Volume III(3), July 1990.

[41] Richard W. Selby, Adam A. Porter, Doug C. Schmidt, and James Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *Proceedings of the 13th International Conference on Software Engineering*, pages 288–298, Austin, Texas, May 1991.

[42] Dick Simmons, Newton Ellis, Hiroko Fujihara, and Way Kuo. *Software Measurement: A Visualization Toolkit*. Prentice-Hall, Inc., 1998.

[43] Koji Torii, Ken ichi Matsumoto, Kumiyo Nakakoji, Yoshihiro Takada, Shingo Takada, and Kazuyuki Shima. Ginger2: An environment for computer-aided empirical software engineering. *IEEE Transactions on Software Engineering*, 25(4), July 1999.

[44] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: A practical guide for quality improvement of software development*. McGraw-Hill, 1999.

[45] D. Webb and Watts Humphrey. Using TSP on the taskview project. *CrossTalk: The Journal of Defense Software Engineering*, February 1999.

[46] Donald J. Wheeler. *Advanced Topics in Statistical Process Control*. SPC Press, 1995.

[47] Donald J. Wheeler and David S. Chambers. *Understanding Statistical Process Control*. SPC Press, 1992.

[48] Marvin Zelkowitz. Establishing a measurement program: Guidelines from the SEL. Course Notes, MSWE 609, 1999.