1 Overview

1.1 Motivation

As with baseball, physics, music, and other skillful human endeavours, there is a vast range of ability associated with software development. For almost 40 years, software development researchers have been attempting to understand, measure, and support the development of superior skill in software development. Sackman performed the seminal research on programmer productivity in 1967, in which he reported a 28:1 difference between the slowest and fastest programmers on a programming task [41]. Subsequent research by Prechelt on Sackman's original dataset in combination with other published datasets indicates a smaller but still significant multiple—from 2:1 to 6:1 depending upon conditions and the kind of statistical comparison used [38]. There is even evidence that some programmers may actually decrease overall productivity, a phenomenon known as the "net negative producing programmer" [43].

While comparison of different individual's effort on a common programming task is the most direct way to measure productivity variability, it is not the only way. One alternative employs the COCOMO II cost estimation model [6]. COCOMO uses a dataset of approximately 160 completed industrial projects to calibrate a model that computes the effort required to complete a project based upon characteristics of the software to be developed and the organization doing the development. In the COCOMO model, the effort differential between best and worst programming teams with respect to capability is 3.53, applications experience is 1.51, language and tools experience is 1.43, platform experience is 1.40, and team cohesion is 1.29. Multiply these together, and the COCOMO model indicates a theoretical productivity difference of 13:1 between the most suited and least suited programming teams for a given software project.

Developer variability creates two basic kinds of challenges for the software engineering research community: (1) How can we raise the average productivity of software developers, and (2) How can we reduce the variability between the best and worst software developers? In general, we have responded to these challenges in one or more of three ways: through abstraction, automation, and through "best" practices.

The evolution of programming languages from machine language to assembly language to high level languages to executable specification languages exemplifies the successful use of abstraction to improve programmer productivity by reducing the amount and complexity of code required to accomplish a given task. A single keyword such as "synchronized" in a high level language like Java might require thousands of lines of code to implement correctly in assembly language. Indeed, software disasters such as the Therac-25 were ultimately attributed to incorrect implementation of process synchronization in application-level software [34].

Automation refers to the development of scripts or other approaches to ensure that a sequence of development tasks are carried out consistently, reliably, and correctly. One example is an automated daily build mechanism, which might (a) create a "clean" initial build state, (b) check out the latest version of a system from a configuration management repository, (c) compile the latest version, (d) deploy the latest version to a run-time environment (such as installation on a web server), (e) run all functional (i.e. unit) and non-functional (i.e. load) tests associated with the latest version, (g) build the documentation associated with the latest version, (h) generate a report associated with the build process, and (i) email results to developers and managers.

The difference between abstraction and automation is that abstraction creates a "black box" for developers while automation does not. For example, the implementation of the synchronized keyword in Java is a black box: no application developer would be expected to maintain or debug this language construct and, in general, developers simply assume that this abstraction functions correctly. A daily build script, however, is typically designed, implemented, and maintained by developers, and thus does not provide abstraction even though it does provide many benefits as a form of automation. For example, it can eliminate the negative productivity impact of developers not carrying out the sequence of actions required to build the product correctly, or even not building the system at all due to the time, overhead, and tedium associated with the activities.

While abstraction is the province of languages and other expressive media, and automation is the province of tools and environments, best practices unifies them with “demonstrably effective” behaviors and activities of people during software development. The seminal software engineering best practice is the waterfall lifecycle model, which was first described in the early 1970’s and provided an efficient and effective partitioning of development into a sequence of phases: specification, design, implementation, testing, and maintenance. Provided that system requirements can be specified in advance and are guaranteed not to change, the waterfall lifecycle model still constitutes a viable best practice for software engineering.

The Software Engineering Body of Knowledge (SWEBOK) illustrates the variety present in the best practices associated with our discipline [1]. SWEBOK provides a map to the state of the art in software engineering, and divides the landscape into ten areas: requirements, design, construction, testing, maintenance, engineering management, configuration management, process, tools, and quality. SWEBOK shows that abstraction, automation, and best practices are not independent concepts but are instead deeply entwined: best practices (such as testing) engender new forms of abstraction (formal languages for testing) and automation (tools for automated test definition and/or invocation). Conversely, new tools (such as automated test frameworks) can catalyze new best practices (such as test driven design).

One might naively assume that becoming a world class software developer would require nothing more than downloading the SWEBOK and implementing all of its best practices and their associated abstractions and automation. Unfortunately, software engineering best practices are highly contextual: a practice that provides immense benefits in one organizational culture and development context might prove disastrous in another. For example, a best practice such as Cleanroom might be essential in the development of a complex, life-critical application but too time-consuming in a startup environment where time to market is critical. To make matters even worse, software engineering best practices can be in direct conflict with each other. The Extreme Programming [4] best practice eschews the use of the Code Inspection [11] best practice, claiming that the use of Pair Programming obviates the need for a separate inspection activity. Given these issues, the term “best practice” is misleading at best: it would be more useful to speak of “preferred practice” (which indicates the practice has been found to be superior to some other practice without implying that the practice is the best possible), or simply “effective practice” (which indicates the practice works without any implications of superiority to any other practice).

The context sensitivity and inconsistency of software engineering best practices (not to mention the misleading nature of the term itself) creates a number of problems. First, how can an organization improve by adoption of best practices when it is so difficult to determine their appropriateness? Some organizations address this problem via a trial-and-error approach, where various best practices are “tried on for size”. Others hire consultants to tell the organization which practices to adopt. Still others utilize models for process improvement such as the CMMI [40], which could be viewed as “best practices for adopting best practices”.

Second, how do “best practices” actually become recognized as such? For example, the best practice of “Extreme Programming” would have likely become a forgotten experiment in an alternative software development process at Chrysler Corporation had Kent Beck not decided to vigorously market the approach with books, lectures, and networking. Ironically, the project on which XP based its initial claims for success was eventually cancelled without fulfilling its requirements and is now used as evidence against XP by its detractors [30].

In summary, software engineering uses three methods to address the problem of programmer productivity variability: abstraction, automation, and best practices. Unfortunately, the creation of best practices with the abstraction and automation they require, and their adoption into new contexts is traditionally mediated by political and social processes that may be quite unrelated to the actual effectiveness of the practice and its associated abstractions/automations in the organization.

1.2 Approach

This research proposal presents a new, continuous, evidence-based approach to the discovery and evaluation of software engineering best practices. Instead of looking outward into the community for the identification and evaluation of best practices for software development, this research will investigate how practices can be identified and evaluated within one's current organizational and project context. Instead of relying on politics or persuasiveness for adoption, this research will show how instrumentation can be used to continuously generate empirical data that provides evidence either for or against the practice. Depending upon the way the evidence is gathered, this research will enable organizations to move beyond the use of the cliche and misleading "best practice" to more refined and useful characterizations, such as "baseline practice" (i.e. this is what we actually do), "effective practice" (i.e. this is what we do and it results in desirable outcomes), or "preferred practice" (i.e. we have compared this approach to others and the evidence suggests that this approach is superior for us).

This approach leverages our research and development activities over the past five years in Project Hackystat [20], an open source framework for continuous, automated collection and analysis of software engineering process and product data. Hackystat implements an automated approach to metrics collection by attaching sensors to development tools. This makes it possible to capture both low and high-level data about processes and products with a combination of precision, completeness, and low overhead not possible with manual approaches. Hackystat also provides a robust implementation of Software Project Telemetry, an approach to in-process monitoring, analysis, and decision-making based upon the generation of high-level abstractions of the sensor data stream. Software Project Telemetry provides a means to understand whether measures of process and produce are stable, improving, or declining over a particular interval in time. In this proposed research, we will augment the Hackystat framework with a new analysis approach called Software Development Stream Analysis (SDSA). The SDSA facility analyzes the low-level behaviors of individuals as they manipulate tools, abstractions, and automation, then applies a rule-based system to characterize the practice or practices in use by the developers. For example, our current prototype implementation of SDSA implements rules that attempt to identify when a developer is using the "test-driven design" practice. Examples of practices that appear to be partially or completely recognizable using the SDSA approach include: configuration management practices such as small check-in sizes; testing practices such as appropriate use of "smoke" tests; compliance to development processes such as PSP, XP, and Scrum; and language choices and their impact on process and product measures.

The combination of Hackystat, Software Project Telemetry, and Software Development Stream Analysis provides a mechanism for continuous, context-sensitive evaluation of practices within an organization. For example, an organization using Hackystat on a project can use Software Project Telemetry to establish baseline values for various software development measures. Software Development Stream Analysis provides a way to identify the use (or non-use) of a practice by the developers. Integrating these techniques provides a way to relate the practices of developers to their outcomes in terms of process and product measures. For example, if a development group decides to adopt the use of pair programming on a trial basis, they can see if this practice makes an impact on the measures of process and product captured by Software Project Telemetry. Conversely, if Software Project Telemetry reveals a significant decline in process or product metrics (such as a drop in the test case coverage of the system), then Software Development Stream Analysis can be used to assess whether some change in practice could be responsible (such as a change from test-first to test-last design).

Our proposed approach contrasts in interesting ways to the more traditional approach to evaluation of best practices. The traditional approach involves the trial adoption of the best practice, the collection of data on the effect of the practice, and an eventual assessment of efficacy of the practice. Trial adoption normally consists of a "one off" experiment on a sample project with specialized data collection during the project, and analysis of the success or failure of the practice once the project is concluded. In contrast, our approach involves the introduction of sensor-based instrumentation into the development environment, which allows continuous, in-process collection and analysis of data concerning the practices in place and their impact on process and product measures. This has significant implications, both positive and negative, for the approach.

On the positive side, the use of low-cost automated metrics collection and analysis allows for “in vivo” rather than “in vitro” analysis of practices: the method is as well suited to characterizing what developers currently do (the “baseline” practice) as what they “should” be doing (the hypothesized “best” practice). As a natural result, it is possible to gain insight into which practice–the baseline or the best–is actually more effective in a specific organizational context. Second, SDSA provides an approach to resolving the problem of process compliance. For example, some development groups claim to be doing “Extreme Programming” while implementing only a fraction of the 12 mandated practices. SDSA can ensure that process and product measures collected using Software Project Telemetry can be accurately related to the practices in place when they were generated. Third, the ongoing presence of instrumentation enables a “bottom-up” approach to best practice discovery, in which the behaviors of successful developers can be more easily analyzed for the presence of repeated patterns, and then compared to process and product-based measures to see if they constitute candidate “best” practices. Moving to the negative side, the fourth implication is that introduction of unobtrusive sensors for “in vivo” analysis of actual practices can lead to legitimate privacy concerns on the part of developers. This can create a barrier to adoption if not addressed. Fifth, care must be taken in industrial contexts to manage the application of practices and assessments to avoid a form of organizational “chaos” where individual development groups diverge wildly in structure and process as they experiment with different practices. The “bottom up” discovery of baseline or preferred practices must be balanced by some amount of “top down” management to ensure that these efforts are consistent and compatible with other business objectives.

1.3 Objectives

The overall objective of this research is to design, implement, and evaluate a continuous, evidence-based approach to discovery and evaluation of best practices during software development. This overall objective has the following sub-objectives:

1. Enhance the prototype Software Development Stream Analysis mechanism to support validation and additional “best” practices. From this we will gain insight into the knowledge engineering required to characterize part or all of a best practice using SDSA, empirically validate the recognition mechanism, and determine the kinds of abstractions, automation, and practices that are amenable to recognition using SDSA.

2. Develop integration mechanisms between SDSA and Software Project Telemetry that allow users to determine how practices recognized by SDSA relate to process and product outcomes.

3. Perform classroom-based evaluation of SDSA and Telemetry. We will apply these techniques to generate evidence regarding programmer productivity and variability with respect to a specific best practice: Test Driven Design. These efforts will also refine the technology, develop curriculum materials, and ready the approach for industrial evaluation.

4. Perform industry-based evaluation of SDSA and Telemetry. Following classroom evaluation, we will carry out two industry-based case studies to gather evidence regarding best practices related to high performance computing and agile software development. In addition to the gathered evidence, this activity will enable us to investigate issues related to privacy and organizational management of this approach in industrial settings.

5. Package the system and methods for widespread dissemination. We will continue the process used by the open source Hackystat Project of making our technology available to the software engineering community. In addition, we will package and disseminate our experimental methods to support external evidence-based software engineering efforts.

6. Develop curriculum materials regarding continuous, evidence-based discovery and assessment of software engineering best practices. As with the Hackystat Project, we will develop software engineering curriculum materials and assignments that enable the study and analysis of this approach in academic settings.

2 Related Work

2.1 Hackystat

For the past five years, we have been developing a framework for automated software development process and product metric collection and analysis called Hackystat. This framework differs from other approaches to software product and process measurement in one or more of the following ways:

- Hackystat uses sensors to unobtrusively collect data from development environment tools; there is no chronic overhead on developers to collect product and process data. In contrast, tools such as the Process Dashboard [39] involve manual data collection.

- Hackystat is tool, environment, process, and application agnostic. The architecture does not suppose a specific operating system platform, a specific integrated development environment, a specific software process, or specific application area. A Hackystat system is configured from a set of modules that determine what tools are supported, what data is collected, and what analyses are run on this data. In contrast, tools such as TSP Tool [10] implement support for a fixed set of metrics under a fixed process on a single platform.

- Hackystat is intended to provide in-process project management support. Traditional software metrics approaches, such as the NASA Metrics Data Program [8], are based upon the "project repository" method, in which data from prior completed projects are used to make predictions about a future project. In contrast, Hackystat is designed to continuously collect data from a current, ongoing project, and use that data as feedback into the current project.

- Hackystat is open source and is available to the academic and commercial software development community for no charge. In contrast, commercial toolkits such as MetricCenter [3] are closed source and require licensing fees.

The design of Hackystat [23] reflects prior research in our lab on software measurement, beginning with research into data quality problems with the PSP [22] and continuing with research on the LEAP system for lightweight, empirical, anti-measurement dysfunction, and portable software measurement [26].

To use Hackystat, the project development environment is instrumented by installing Hackystat sensors, which developers attach to the various tools such as their editor, build system, configuration management system, and so forth. Once installed, the Hackystat sensors unobtrusively monitor development activities and send process and product data to a centralized web server. If a user is working offline, sensor data is written to a local log file to be sent when connectivity can be re-established. Project members can log in to the web server to see the collected raw data and run analyses that integrate and abstract the raw sensor data streams into telemetry. Hackystat also allows project members to configure "alerts" that watch for specific conditions in the sensor data stream and send email when these conditions occur. Figure 1 illustrates the basic architecture of the system.

Hackystat is an open source project. Its sources, binaries, and documentation are freely available online. We also maintain a public server running the latest release of the system at http://hackystat.ics.hawaii.edu. Hackystat has been under active development for approximately five years, and currently consists of approximately 2500 classes and 300,000 lines of code. Sensors are available for a variety of tools including Eclipse, Emacs, JBuilder, Jupiter, Jira, Visual Studio, Ant, JUnit, JBlanket, CCCC, DependencyFinder, Harvest, LOCC, Office, CVS, and SVN.

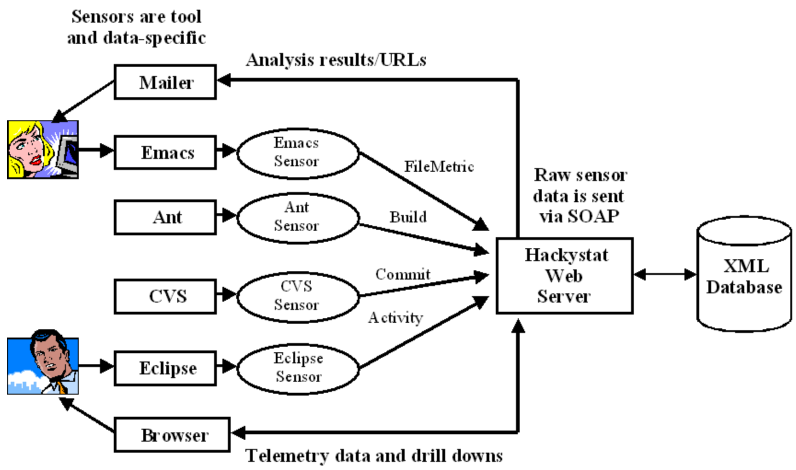

Figure 1. The basic architecture of Hackystat.

Hackystat is being used in a variety of academic and industrial contexts. At the University of Hawaii, Hackystat has been integrated into the undergraduate and graduate software engineering curriculum, and is used by approximately 50 students per year to support project development [24]. A researcher from the Free University of Bozen came to Hawaii to study the Hackystat system in support their research on PROM [45]. Researchers at the University of Maryland are using Hackystat to support assessment of programmer effort [15]. Hackystat has been used at NASA's Jet Propulsion Lab to analyze the daily build process for the Mission Data System [21]. Finally, Hackystat is being used at SUN Microsystems to support research on high performance computing system development productivity [12].

2.2 Software Project Telemetry

The automated, unobtrusive, continuous, and low-cost measurement infrastructure provided by Hackystat enabled us to develop a new approach to software measurement analysis called "Software Project Telemetry". According to Encyclopedia Brittanica, telemetry is a "highly automated communications process by which measurements are made and other data collected at remote or inaccessible points and transmitted to receiving equipment for monitoring, display, and recording." We define Software Project Telemetry as a style of software engineering process and product collection and analysis which satisfies the following five properties:

(1) Software project telemetry data is collected automatically by tools that unobtrusively monitor some form of state in the project development environment. In other words, the software developers are working in a "remote or inaccessible location" from the perspective of metrics collection activities. This contrasts with software metrics data that requires human intervention or developer effort to collect, such as PSP/TSP metrics [16].

(2) Software project telemetry data consists of a stream of time-stamped events, where the time-stamp is significant for analysis. Software project telemetry data is thus focused on evolutionary processes in development. This contrasts, for example, with COCOMO [6], where the moment in time at which calibration data is collected is not generally significant.

(3) Software project telemetry data is continuously and immediately available to both developers and managers. Telemetry data is not hidden away in some obscure database guarded by the software quality improvement group. It is easily visible to all members of the project for interpretation.

(4) Software project telemetry exhibits graceful degradation. While complete telemetry data provides the best support for project management, the analyses should not be brittle: they should still provide value even if sensor data occasionally "drops out" during the project. Telemetry collection and analysis should provide decision-

making value even if these activities start midway through a project.

(5) Software project telemetry is used for in-process monitoring, control, and short-term prediction. Telemetry analyses provide representations of current project state and how it is changing at the time scales of days, weeks, or months. The simultaneous display of multiple project state values and how they change over the same time periods allow opportunistic analyses—the emergent knowledge that one state variable appears to co-vary with another in the context of the current project.

Software Project Telemetry enables an incremental, distributed, visible, and experiential approach to project decision-making. For example, if one finds that complexity telemetry values are increasing, *and* that defect density telemetry values are also increasing, then one could try corrective action (such as simplification of overly complex modules) and see if that results in a decrease in defect density telemetry values. One can also monitor other telemetry data to see if such simplification has unintended side-effects (such as performance degradation). Project management using telemetry thus involves cycles of hypothesis generation (Does module complexity correlate with defect density?), hypothesis testing (If I reduce module complexity, then will defect density decrease?), and impact analysis (Do the process changes required to reduce module complexity produce unintended side-effects?). Finally, Software Project Telemetry supports decentralized project management: since telemetry data is visible to all members of the project, it enables all members of the project–developers and managers–to engage in these management activities.

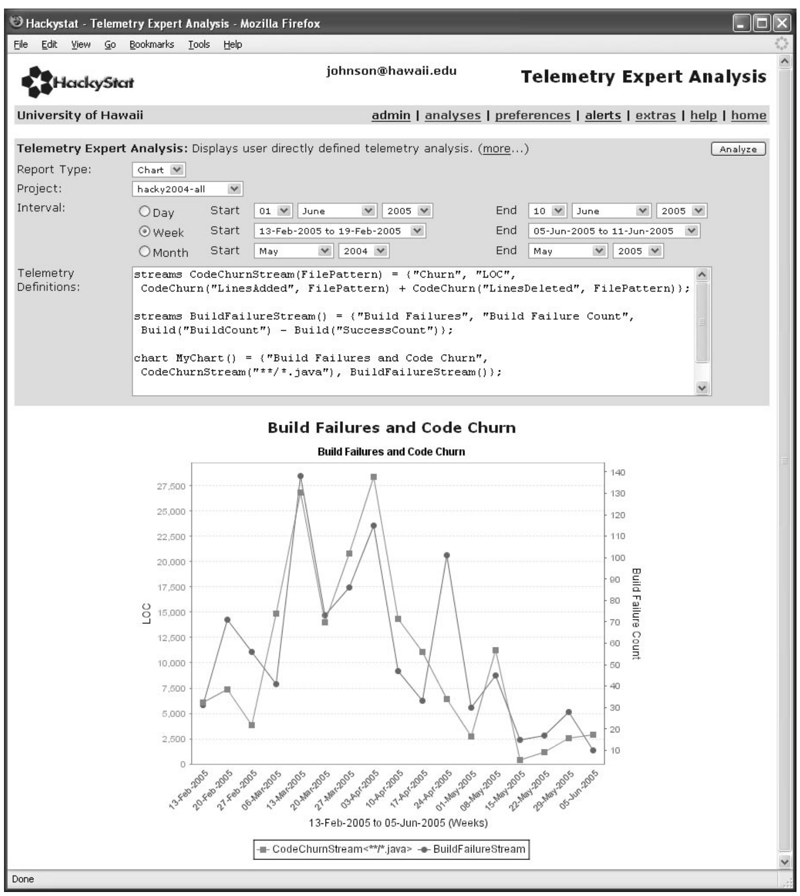

Figure 2. Telemetry illustrating how code churn co-varies with build failures.

As a concrete example of telemetry, consider Figure 2. This report illustrates the relationship between aggregate code churn (the lines added and deleted from the CVS repository by all members of the project) and the number of build failures over a four month period on the Hackystat project. Note how closely these two measures co-vary, even though one is a process measure (build failure) and the other is a product measure (code churn). From this initial observation, one could investigate other time periods and time scales to see if this relationship holds in other contexts, as well as test hypotheses on how to reduce build failures or predict their impact on the project schedule.

The computational path from the sensors in Figure 1 to the telemetry report in Figure 2 involves several steps. In the first step, "raw" sensor data is collected from small software plug-ins attached to developer tools. For example, an editor sensor may record a "state change" event when a file has been edited within the last 30 seconds by the user. A CVS sensor may record the number of lines added and deleted from a file during the past 24 hours. This raw data is sent from the sensors to a Hackystat server, where they are persisted in an XML-based repository. In the second step, the system abstracts the raw sensor data into one or more "DailyProjectData" instances, which synthesize raw sensor data from multiple group members and/or multiple sensors into a higher level abstraction. For example, a DailyProjectData instance might process low-level "state change" events from multiple developers and determine the total amount of time spent editing files by all members of the project group for a given day. In the third step, special classes called "Reduction Functions" manipulate DailyProjectData instances to create the sequence of numerical telemetry values associated with a given project and time interval. For example, a Reduction Function might manipulate a set of DailyProjectData instances to produce a sequence of numerical telemetry values indicating LOC/hour. In the fourth step, the developer defines a Telemetry Chart or Report, which specifies what types of product and process data should be displayed together. Finally, the Telemetry Chart or Report is instantiated for a specific Project and time interval.

In one application of Software Project Telemetry, we are creating telemetry streams to support diagnosis of daily build failures and reduce the productivity impact of their occurrence over time [25]. Another application involves the development of specialized telemetry streams for high performance computing software to better understand the bottlenecks present in the development of those systems [27].

2.3 Evidence-based software engineering

A recent revolution in medical research involves the introduction of an "evidence-based" paradigm. This paradigm arose in response to two observations: the failure to organize medical research into systematic reviews could cost lives, and the clinical judgement of experts compared unfavorably with the results of systematic reviews. The evidence-based approach is starting to be applied outside of medicine, in fields such as psychiatry, nursing, social policy, education, and software engineering.

Kitchenham has been leading the movement for evidence-based software engineering, organizing workshops on this topic and publishing papers explaining the issues involved in applying evidence-based research techniques to software engineering [32, 31]. She and her collaborators propose a five step method for evidence-based software engineering: (1) Convert the need for information [about a software engineering practice] into an answerable question; (2) Track down the best evidence available for answering the question; (3) Critically appraise that evidence using systematic review for its validity (closeness to the truth), impact (size of the effect), and applicability (usefulness in software development practice); (4) Integrate the critical appraisal with current software engineering knowledge and stakeholder values [to support decision-making]; (5) Evaluate the effectiveness and efficiency in applying Steps 1-4 and seek ways to improve them for next time. While promising, application of systematic reviews and the integration of empirical software engineering data from multiple sources has been found to be challenging [18].

2.4 Software process research

Osterweil has developed a view of software process research that recognizes two complementary levels: macroprocess and microprocess [36]. Macroprocess research is focused on the outward manifestations of process—the time taken, costs incurred, defects generated, and so forth. Macroprocess research traditionally correlates such outcome measures to other project characteristics, which can suggest the impact of process changes to these outcomes, but which suffers from the lack of any underlying causal theory. Bridging this gap is the province of microprocess research, according to Osterweil, in which languages and formal notations are used to specify process details at a sufficient level of rigor and precision that they can be used to support causal explanation of the outcome measures observed at the macroprocess level. In most cases, these forms of software process research are "top-down", in that the goal is to specify a "best practice" in abstract terms and in such a way that it can be enforced within the development environment [44, 17, 35, 7, 42, 46]. Our research most readily fits into the "microprocess" level, except that instead of specifying a top-down language, our approach focuses on bottom-up recognition of the "actual" process.

The Balboa research project, like our proposed research, is concerned with bottom-up inference of process from low-level event streams [9]. In Balboa, the event streams are taken from the commit records of a configuration management system, and finite state machines are created that model the commit stream data observed in practice. More recently, work has been done on understanding processes associated with open source software development processes [19]. In this research, "web information spaces" are mined with the goal of discovering software process workflows via analysis of their content, structure, update, and usage patterns. Our approach. Our proposed approach contrasts with these by employing sensor instrumentation attached to a broad variety of developer tools including their interactive development environment. This enables our analysis mechanisms access to much lower-level events than those available through the commit records of a configuration management system or through web-based information sources.

2.5 Results from prior NSF research

Award number:	CCF02-34568
Program:	Highly Dependable Computing and Communication Systems Research
Amount:	$638,000
Period of support:	September 2002 to September 2007
Title of Project:	Supporting development of highly dependable software through continuous, automated, in-process, and individualized software measurement validation
Principal Investigator:	Philip M. Johnson
Selected Publications:	[27, 37, 25, 24, 23, 21, 29, 13, 28]

The general objective of this research project is to design, implement, and validate software measures within a development infrastructure that supports the development of highly dependable software systems. Contributions of this research project include: (a) development of a specialized configuration of Hackystat to automatically acquire build and workflow data from the configuration management system for the Mission Data System (MDS) project at Jet Propulsion Laboratory; (b) development of analyses over MDS build and workflow data to support identification of potential bottlenecks and process validation; (c) identification of previous unknown variation within the MDS development process; (d) development of a generalized approach to in-process, continuous measurement validation called Software Project Telemetry, (e) substantial enhancements to the open source Hackystat framework, improving its generality and usability; (f) development of undergraduate and graduate software engineering curriculum involving the use of Hackystat for automated software engineering metrics collection and analysis; (g) support for 3 Ph.D., 6 M.S., and 3 B.S. degree students.

3 Research Plan

To evaluate the feasibility of this research approach, we have already implemented a prototype version of SDSA, augmented it with rules for the Test Driven Development best practice, and performed a pre-pilot study on a small set of students. These preliminary results are encouraging and suggest that SDSA does have the potential to recognize at least one non-trivial software engineering best practice. However, significant research questions remain: will these preliminary results hold up when applied to more sophisticated users of TDD? What are effective approaches to validation of the recognition rules? What are the limitations of the Hackystat sensor-based technology for gathering the raw data necessary for best practice recognition? Can SDSA still be useful when only "partial" recognition is possible? Can our prototype single user version of SDSA scale to recognition of behavioral patterns across groups of developers? Finally, what is the most effective way to design, package, and distribute the technology to support replication, adoption, and enhancement to support new practices by other organizations?

To provide insight into these research questions and our research task, the next section provides more details on our proposed analysis mechanism.

3.1 Software Development Stream Analysis

Software Project Telemetry supports a "macro" view of project development: telemetry aggregates process and product data gathered by the developers in a project and how they change at time scales of days, weeks, or months. In contrast, our proposed Software Development Stream Analysis mechanism focuses on a "micro" view of project development, which seeks to analyze the behaviors of a single developer at the time scale of minutes or hours and characterize the practices, if any, being employed by the developer.

To make this concrete, consider the best practice called "test driven design" (TDD) [5]. TDD is often explained through a stop light metaphor, which cycles from green to yellow to red. At the beginning of a TDD cycle, the code is working, and the stop light is green. The developer then defines a new test case that tests a new (and as yet unimplemented) feature. This will produce a syntax (compilation) error because that feature is not even implemented yet. This changes the stop light to yellow. Consequently, the developer implements a stub version of the feature, which fixes the compilation error but produces a test failure. This changes the stop light to red. Once the developer finishes implementing the feature, the light changes to green and the cycle begins again. The stop light metaphor is interesting because out of sequence lights indicate violations of the TDD development pattern; for example, green to red indicates that the developer added new code without adding a test for it first.

Recognizing practices such as TDD using SDSA involves a multi-step process. First, Hackystat sensors collect raw data sufficient to allow identification of the development behavior of interest. In the case of a practice like TDD, only a sensor for the IDE is required, and it collects editing events, compilation events, refactoring events, and test invocation events. These events are sent to the Hackystat server for further analysis. On the server side, the second step involves "tokenization", which essentially replaces sequences of raw data representing the same type of behavior by a single token representing that behavior. For example, if the user edits the same file for several minutes, several dozen editing events may be generated in the raw data stream. The SDSA tokenizer replaces these by a single "Edit" token. Third, SDSA applies rules for partitioning the single tokenized event stream into a sequence of "episodes", where each episode represents a behavioral sequence that can be recognized as belonging to the practice or not. In the case of the TDD best practice, an appropriate episode boundary is when all of the unit tests pass. Fourth, SDSA applies classification rules based upon the open source JESS framework [14] to identify each episode as belonging to the best practice or not. For example, an episode belonging to the TDD practice might consists of a sequence of tokens indicating: (a) creation of a test case; (b) a failed attempt to compile the test case; (c) editing (presumably to fix the compilation error by implementing the production code invoked in the test case); (d) invocation and failure of the test case (presumably due to a bug in the implementation); (e) further editing; and (f) successful test case invocation (which also signals the episode boundary).

This all sounds good in theory, but does it actually work in practice? Can sensors actually capture the raw data necessary for higher-level analyses? Can appropriate episode boundaries be defined? Finally, does the model of the best practice as defined by the recognition rules correspond to the reality of the best practice as understood by practitioners?

To provide an initial feasibility check of SDSA, we performed a pre-pilot study [33]. For this study, we created a package called Zorro that extends the generic SDSA mechanism with the episode recognition and classification rules for TDD. We also implemented a system called "Eclipse Screen Recorder" (ESR), which creates a QuickTime movie of the Eclipse window. We then asked seven volunteers to develop a simple application in Eclipse while following basic Test Driven Design practices. The participants completed the task in a single programming session that typically lasted between 30 and 60 minutes. We captured their development behaviors in two independent ways: through the sensor events collected by Hackystat as well as though the QuickTime movie created using ESR. Using the QuickTime movie, we performed a validation of the Zorro TDD recognition package, checking that the movie of developer behaviors corresponded to the behavioral patterns inferred by Zorro. Our results revealed opportunities for improvement in both sensor data collection and rule definition, but despite these problems we found that Zorro correctly classified 89% of the 92 episodes generated in this study.

The success of our pre-pilot study gives us confidence that SDSA is a viable research direction and reduces some of the risk factors associated with this project.

3.2 Task descriptions

Our research plan follows the detailed objectives for this research as summarized in Section 1.3, and consists of the following tasks.

(1) Enhance the Software Development Stream Analysis mechanism. There are two primary kinds of enhancements that we propose in this research. The first involves more sophisticated support for validation. In our pre-pilot research, we used the Eclipse Screen Recorder to obtain an independent view of developer behaviors. One limitation of this technique is that it relies on the experimenters own definition of TDD. To overcome this limitation, we would like to provide users with the ability to review and confirm the inferences made by the system, which can provide valuable feedback about the practice as understood by the practitioners themselves.

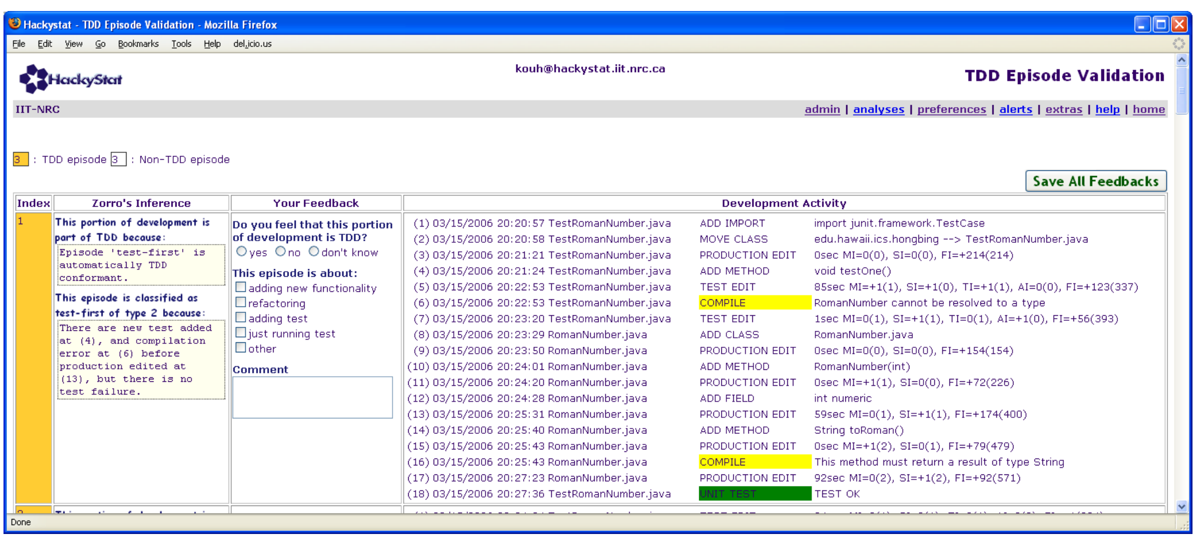

Figure 3. A mockup of an enhanced SDSA validation interface.

Figure 3 illustrates a mockup of an interface for TDD validation. Each row in this interface represents an episode. The interface allows the user to provide feedback about the classification of each episode after review of the classification made and the reasons behind it (which are automatically generated by traversing the rule base).

A second task associated with this objective is to expand the range of practices supported by SDSA in order to better understand its strengths and limitations. We intend to expand support for agile programming practices beyond TDD to pair programming and continuous integration.

In addition, we plan to extend SDSA to support high performance computing best practices, based upon our association with the DARPA High Productivity Computing Systems (HPCS) program. One of the workflow models identified in this program separates high performance computing development activities into serial coding, parallel coding, debugging, and optimization. However, there is debate within the community regarding the way these workflow activities should be organized: some advocate serial coding first (in order to establish functional correctness without the complexity of parallelization), while others advocate parallel coding first (to establish a parallel framework within which to embed the serial code). There is even debate as to whether this workflow decomposition is the appropriate way to characterize HPC development. Research on an SDSA-based recognition mechanism for HPC activities can both provide empirical data regarding these debates, as well as help us understand the strengths and limitations of the SDSA approach.

(2) Develop integration mechanisms between SDSA and Software Project Telemetry. Currently, Hackystat does not provide any support for relating the analyses of SDSA to those of Software Project Telemetry. In this task, we will develop such mechanisms. In general, the goal is to support the display of relationships in two directions. Given the identification of an interesting SDSA behavior, we will display potentially relevant telemetry streams in the period surrounding that behavior. Conversely, given the identification of an interesting change in the telemetry associated with a given project, we will display the SDSA behaviors in the period surrounding that change.

We have already found that in some cases, it is possible to convert the SDSA results into telemetry streams, greatly simplifying the integration. For example, the Zorro system provides a telemetry stream representing the percentage of episodes during the day recognized as TDD-compliant. This makes a number of interesting telemetry-based analyses easily available, such as one that shows how test case coverage varies with the percentage of TDD-compliant episodes during development over time. Given that TDD is purported to achieve 100% test coverage as a natural by-product of the practice, this telemetry can help verify that claim, and also see if "graceful degradation" in coverage levels occurs if the percentage of TDD-compliant episodes goes below 100%.

(3) Perform classroom-based, case study evaluation of the proposed techniques. Once we have implementations of SDSA and its Telemetry integration, we will perform studies to assess their utility and effectiveness in classroom conditions. Over the past several years, we have integrated Hackystat into the undergraduate and graduate software engineering curriculum at the University of Hawaii. Our students now routinely install sensors, collect and analyze process and product metrics, and use this data to guide project management [24]. This provides an excellent environment for initial evaluation of new Hackystat-based technologies.

We plan to evaluate SDSA and Telemetry integration though a case study of test-driven design with three experimental phases. In the first phase at the beginning of the course, we will introduce "traditional" unit testing and have them carry out a project assignment that requires them to achieve at least 90% coverage (to guarantee a minimal level of test case quality) but without specifying when or how to develop the test cases. We will use SDSA to verify that students are not using TDD for this phase, and use Software Telemetry to assess coverage, LOC/Hour, and other indicators of quality and productivity.

In the second phase of the experiment, we will introduce principles of TDD, provide a short assignment for them to use to learn TDD, then have them carry out a project assignment that requires both the use of TDD and at least 90% coverage.We will use SDSA to verify that the students are using TDD during this project, and Software Telemetry to assess the same indicators of quality and productivity as before. We can compare the values of these indicators to see if any changes occur from the introduction of TDD. For example, does the introduction of the TDD best practice improve average programming productivity? Does it decrease the variation in programmer productivity? Finally, we will use validation facilities like the mockup illustrated in Figure 3 to obtain developer opinions on classification accuracy.

In the third phase of the experiment, we will have the students carry out a final project assignment, and this time require 90% coverage but allow them to choose their test development process. We will use SDSA to determine what percentage of the students use TDD and how consistently they use TDD. We will use telemetry to see how quality and productivity measures have changed relative to earlier phases.

At the end of the study, we will collect qualitative data using a questionnaire to assess student attitudes towards Hackystat, TDD, and SDSA.

We foresee multiple uses for the results from this case study. First, it will provide useful information about the robustness and utility of our technology. For example, can SDSA correctly characterize developer behavior as TDD, and to what extent is it susceptible to false positives and false negatives? In addition, the case study data can yield interesting empirical evidence regarding the efficacy of TDD, its impact on programmer productivity and variability, and provide an empirical, replicable test of the claims made by its proponents.

(4) Perform industry-based evaluation of SDSA and Telemetry. Once our classroom evaluation is under way and we feel confident that the technology and methods are sound, we plan to carry out two industrial case studies.

The first industry case study will involve programmers affiliated with the DARPA High Productivity Computing Systems program. In this case study, we will identify a high performance system software development group interested in obtaining empirical data regarding their HPC development practices, install the enhancements described above in (1), monitor usage, and collect validation data to assess the accuracy of the inferences. We anticipate that the size of this development group will be on the order of 6-10 developers, and that we will monitor their practices over a period of approximately six months.

The second industry case study will use SDSA to assess test-driven development in an industrial setting. We will publish a call for an "open evaluation" of agile practices at such conferences as XP/Agile Universe and internet sites such as the Agile Alliance. We hope to attract approximately 50 participants who will agree to have their agile development practices monitored for a period of one week. Participants will download and install the appropriate sensors and begin sending data to the Hackystat public server. We will use the enhanced version of SDSA as discussed above to determine when they are performing agile activities such as TDD, pair programming, and continuous integration. The server will provide them with analyses of their data, and include validation mechanisms to enable them to report on the correctness of the classification mechanisms. At the conclusion of the open evaluation, we will distribute a questionnaire to participants to collect data on their experience and demographic information that we can use to better understand the context in which their data was generated.

The industrial case study results will be used to assess the robustness and utility of the Telemetry and SDSA mechanisms outside of a controlled classroom setting. In addition, the case studies will produce new empirical evidence regarding TDD and HPC best practices. Finally, it will provide insights into the limitations of this research approach, including (a) the extent to which automated sensor-based technology and rule-based inference is adequate to recognize best practices in industrial settings, and (b) the privacy and organizational barriers to successful adoption of this approach.

(5) Package the system and methods for widespread dissemination. Our approach to this task is highly influenced by the movement toward evidence-based software engineering, which seeks the introduction of systematic reviews to improve the quality of our understanding of practices, the increased use of replication to better understand the context surrounding empirical results, and a better understanding of how evidence can be used to support the practice of software engineering in real world contexts.

As Hackystat and its associated applications are open source software with a well-developed infrastructure for distributed development, the packaging of the actual software for widespread dissemination is straightforward. A more challenging problem is to package the experimental methods such that the software can be used effectively to produce empirical evidence that adds new value to a systematic review of the literature, either via replication of an existing experiment or via a modification to an existing method. We plan to build upon prior research by Basili and his colleagues on "Experience Factories" [2] to create "kits" combining a Hackystat software configuration with documentation detailing the sensors to install, the data to collect, and the analyses to perform to gain insight into the best practice of interest.

(6) Develop curriculum materials. Our prior experience with Hackystat in the classroom setting convinces us that collection and analysis of software engineering metrics can form a compelling motivation for the use of software engineering practices such as unit testing, configuration management, and software review. For this task, we will incorporate the technology and methods from this evidence-based approach to best practices into our undergraduate and graduate software engineering curriculum. We will teach the theory of evidence-based software engineering, show data from prior studies and discuss its implications, and most importantly, enable students to experience the gathering and analysis of evidence about their own practice through the use of Hackystat, SDSA, and Software Project Telemetry on classroom projects.

3.3 Work breakdown structure and milestones

Tasks and Milestones	Fall 07	Spr 08	Fall 08	Spr 09	Fall 09	Spr 10
(1) SDSA Enhancement						
(2) SDSA/Telemetry integration						
Milestone: Release 8.0		X				
(3) Classroom case study						
(4) Industry case studies						
Milestone: Release 9.0				X		
(5) Packaging and dissemination						
(6) Curriculum materials						
Milestone: Release 10.0						X

Figure 4. Work breakdown structure and milestones.

Figure 4 shows when we plan to carry out each of these tasks, along with three milestones. As illustrated, we will be working on SDSA enhancement for almost the entire time of the project. The case studies will begin in year two, and the packaging and curriculum material generation will be the primary focus of year three. We also plan for three major milestones during the project, occurring near the end of each of the three years of the research. The milestones are denoted by upcoming major releases of the Hackystat framework and its associated applications. Release 8.0 will include the enhanced software development stream analysis mechanism, along with integration mechanisms for software project telemetry. Release 9.0 will incorporate enhancements based upon the classroom studies. At the end of the project, we will release 10.0, which enhances the prior releases with the all of the results from the project.

It is also important to identify and manage the risk factors associated with this research plan. We view the enhancement of SDSA as having moderate risk, in that we do not yet know how well the current SDSA design generalizes beyond the case of test-driven design. It may be that substantial redesign will be required to generalize our approach to best practice representation beyond TDD. The SDSA/Telemetry integration task has low risk: this is an engineering modification to the Hackystat framework that we are well suited to accomplishing. The classroom case studies have low risk: we have a great deal of experience with classroom case studies and have ready access to the software engineering student population at the University of Hawaii. The industry case studies have moderate risk; while we have an excellent collaborative relationship with the DARPA HPCS participants, there are always political and organizational hurdles to cross before a case study can occur on a real-world project. The TDD case studies include the risk of failing to attract interest from the Agile development community. Finally, the packaging, dissemination, and curriculum material development tasks have low risk; we have been developing curriculum materials and packaging/disseminating our Hackystat research for a number of years and are experienced with this activity.

4 Conclusions

In all fields of human endeavor, there are extraordinarily gifted practitioners: in music, Beethoven and Mozart; in golf, Wie and Woods; in software development, Stallman and Joy. While no technological innovation can match innate genius, the ultimate goal of this research is to provide the tools and techniques to allow all software developers to reach their maximal potential.

Our proposal presents research designed to produce a variety of contributions to the theory and practice of software engineering as well as broader impact to society at large. First, it will yield new technological infrastructure for continuously collecting, analyzing, and interpreting software engineering best practices. This infrastructure will be novel in its ability to collect and analyze both “macro” level project characteristics and “micro” level developer practices and relate these two levels of information to each other.

Second, the research will yield a set of case studies in both classroom and industrial settings. The studies are designed to provide new empirical data about software engineering, new insights into how best practices and be represented and used, and specific data about programmer productivity and behavior in domains including high performance computing and agile development. It will also provide findings regarding the limitations of this approach due to technical or organizational factors.

Third, the research is grounded in an evidence-based approach to software engineering, which should yield results more easily available to systematic review, replication, and enhancement. We intend for our curriculum materials to be leveraged by other teachers and result in improved use of metrics in software engineering practice. As the University of Hawaii is a university with 75% minority students in an EPSCOR state, this research will provide novel research opportunities to underrepresented groups.

There was a sign that hung in Albert Einstein’s office at Princeton University: “Not everything that counts can be counted, and not everything that can be counted counts”. This cautionary statement is certainly relevent to our proposed research. However, we believe that this combination of contributions, if successful, will provide one more step toward safer, sounder, and more cost-effective information technology for our society.

References

[1] Alain Abran and James Moore, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2005.

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

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

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[5] Kent Beck. *Test-Driven Development by Example*. Addison Wesley, 2003.

[6] 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.

[7] Aaron Cass, Barbara Lerner, Eric McCall, Leon Osterweil, Stanley Sutton, and Alexander Wise. Little-jil/juliette: A process definition language and interpreter. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[8] Mike Chapman. NASA MDP repository. http://mdp.ivv.nasa.gov/, 2004.

[9] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

[10] Noopur Davis. Team Software Process tool. http://www.sei.cmu.edu/tsp, 2004.

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

[12] Stuart Faulk, John Gustafson, Philip M. Johnson, Adam A. Porter, Walter Tichy, and Larry Votta. Toward accurate HPC productivity measurement. In *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 2004.

[13] Stuart Faulk, Philip M. Johnson, John Gustafson, Adam A. Porter, Walter Tichy, and Larry Votta. Measuring HPC productivity. *International Journal of High Performance Computing Applications*, December 2004.

[14] Ernest Friedman-Hill. *JESS in Action*. Mannig Publications Co., Greenwich, CT, 2003.

[15] Lorin Hochstein, Victor Basili, Marvin Zelkowitz, Jeffrey Hollingsworth, and Jeff Carver. Combining self-reported and automatic data to improve effort measurement. In *Proceedings of the 2005 Conference on Foundations of Software Engineering*, 2005.

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

[17] Dirk Jager, Ansgar Schleicher, and Bernhard Westfechtel. Using UML for software process modeling. 1999.

[18] A. Jedlitschka and M. Ciolkowski. Towards evidence in software engineering. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, 2004.

[19] Chris Jensen and Walt Scacchi. Experience in discovering, modeling, and reenacting open source software development processes. In *Proceedings of the International Software Process Workshop*, 2005.

[20] Philip M. Johnson. Hackystat Framework Home Page. http://www.hackystat.org/.

[21] Philip M. Johnson. The Hackystat-JPL configuration: Overview and initial results. Technical Report CSDL-03-07, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, October 2003.

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

[23] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Christopher Chan, Carleton A. Moore, Jitender Miglani, Shenyan Zhen, and William E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.

[24] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, Los Angeles, California, August 2004.

[25] Philip M. Johnson, Hongbing Kou, Michael G. Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Improving software development management through software project telemetry. *IEEE Software*, August 2005.

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

[27] Philip M. Johnson and Michael G. Paulding. Understanding HPCS development through automated process and product measurement with Hackystat. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, February 2005.

[28] Aaron Kagawa. Hackystat MDS supporting MSL MMR. Technical Report CSDL-04-06, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, June 2004.

[29] Aaron Kagawa and Philip M. Johnson. The Hackystat-JPL configuration: Round 2 results. Technical Report CSDL-03-07, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, May 2004.

[30] Gerold Keefer. Extreme programming considered harmful for reliable software development. Technical report, AVOCA GmbH, 2003.

[31] B. Kitchenham. Systematic reviews. In *Proceedings of the 2004 International Symposium on Software Metrics*, 2004.

[32] Barbara Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *Proceedings of the 2004 International Conference on Software Engineering*, 2004.

[33] Hongbing Kou and Philip M. Johnson. Automated recognition of low-level process: A pilot validation study of Zorro for test-driven development. In *Proceedings of the 2006 International Workshop on Software Process*, Shanghai, China, May 2006.

[34] Nancy Leveson and Clark Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, July 1993.

[35] Elisabetta Di Nitto, Luigi Lavazza, Marco Shiavoni, Emma Trananella, and Michelle Tombetta. Deriving executable process descriptions from UML. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[36] Leon J. Osterweil. Unifying microprocess and macroprocess research. In *Proceedings of the International Software Process Workshop*, pages 68–74, 2005.

[37] Michael G. Paulding. Measuring the processes and products of HPCS development: Initial results for the optimal truss purpose-based benchmark. Technical Report CSDL-04-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, September 2004.

[38] Lutz Prechelt. The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical Report 1999-18, University of Karlsruhe, 1999.

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

[40] Walker Royce. CMM vs. CMMI: From conventional to modern software management. *The Rational Edge*, February 2002.

[41] H. Sackman, W. Erikson, and E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1), 1968.

[42] Craig Schlenoff, Michael Gruninger, Florence Tissot, John Valois, Josh Lubell, and Jintae Lee. *The Process Specification Language (PSL) Overview and Version 1.0 Specification*. National Institute of Standards and Technology, 2000.

[43] G. Schulmeyer. The net negative producing programmer. *American Programmer*, June 1992.

[44] Terry Shepard, Steve Sibbald, and Colin Wortley. A visual software process language. *Communications of the ACM*, April 1992.

[45] Alberto Sillitti, Andrea Janes, Giancarlo Succi, and Tullio Vernazza. Collecting, integrating and analyzing software metrics and personal software process data. In *Proceedings of the 29th Euromicro Conference*, 2003.

[46] Stanley Sutton and Leon Osterweil. The design of a next-generation process language. In *Proceedings of the Fifth International Symposium on Foundations of Software Engineering*, 1997.