Improving Software Development Management through Software Project Telemetry

Philip M. Johnson Hongbing Kou Michael Paulding
Qin Zhang Aaron Kagawa Takuya Yamashita
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu

Abstract

Software project telemetry is a new approach to software project management in which sensors are attached to development environment tools to unobtrusively monitor the process and products of development. This sensor data is abstracted into high-level perspectives on development trends called Telemetry Reports, which provide project members with insights useful for local, in-process decision making. This paper presents the essential characteristics of software project telemetry, contrasts it to other approaches such as predictive models based upon historical software project data, describes a reference framework implementation of software project telemetry called Hackystat, and presents our lessons learned so far.

1. Introduction

It is conventional wisdom in the software engineering research community that metrics can improve the effectiveness of project management. Proponents of software metrics quote theorists and practitioners from Galileo's "What is not measurable, make measurable" [5] to DeMarco's "You can neither predict nor control what you cannot measure" [3]. Software metrics range from internal product attributes, such as size, complexity, and modularity, to external process attributes, such as effort, productivity, testing quality, and reliability [4].

Despite the potential of metrics in theory, effectively applying them appears to be far from mainstream in practice. For example, a recent case study of over 600 software professionals revealed that only 27% viewed metrics as "very" or "extremely" important to their software project decision making process [8]. The study also revealed that cost and schedule estimation was the only use of metrics attempted by a majority of respondents.

At least two hypotheses could account for the gulf separating the theory from the practice of software metrics. The first is that three quarters of practitioners are just plain uninformed: if they would subscribe to the journals and/or partake of the many educational opportunities regarding software metrics, they would immediately implement current "best practice" that would just as quickly improve their project management decision making.

While all of us, theorists and practitioners alike, can always benefit from additional reading and education, there is an alternative explanation. Perhaps the metrics methodology used by theorists yields results not easily translated into practice? Consider that much metrics research involves the following basic methodology: (1) collect a set of process and product measures (such as size, effort, known defects, complexity, etc.) for a set of completed software projects; (2) generate a model that fits this data; (3) claim that this model can now be used to predict characteristics of future projects. For example, a model might predict that a future project of size S will require E person-months of effort; another that the future implementation of a module with complexity C will be prone to defects with density D.

Practitioners face several barriers to adoption of these predictive, model-based approaches to metrics. First, to use the model unchanged, practitioners must confirm that the set of projects used to calibrate the model are similar to their own. This is the *Context Problem*: unless the context associated with the software process and project data in the repository is similar to the context associated with the practitioner's projects, then the applicability of the model's outputs to the practitioner is in question. Similarly, practitioners must also confirm that the context of their future projects will remain similar to their previous ones. If those two conditions cannot be met, then the model cannot be used without recalibration. This involves a replicating the model-building methodology within the practitioner's organization, with the risks that the resulting model will not work or that the organization may change, rendering the context of future projects different from those used to calibrate the model.

Faced with these barriers, it is no wonder that many practitioners find it daunting to apply metrics best practices to their own situation. Indeed, the agile community generally argues against model-based metrics applications, promoting much "softer" metrics for decision-making [1].

Fortunately, creation of predictive models based upon historical project data is not the only possible way to apply software metrics to project management. In this paper, we present a new approach based upon the notion of "telemetry".

2. 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." Perhaps the highest profile user of telemetry is NASA, where telemetry has been used since 1965 to monitor the early Gemini missions to the modern Mars rover flights. At NASA's Mission Control Center, for example, dozens of specialists monitor telemetry data sent from sensors attached to a space vehicle and its occupants. This data is used for many purposes, including early warning of anomalies indicating problems, for better insight into the current status of the mission, and for the impact of making incremental course or mission adjustments.

We define "software project telemetry" as a style of software metrics definition, collection, and analysis with the following essential 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 inaccessable 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 [6].

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 [2], where the time at which the calibration data was collected about the project is not 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 a more 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.

Software Project Telemetry is related to in-process software metrics, such as work done on management of software testing [7]. However, such work tends to focus on a narrow range of measures and management actions, and as as a result may involve manual collection and/or analysis of data. The broader scope of telemetry necessitates automated collection and analysis, with a corresponding broader range of management decision-making support.

3. Supporting Software Project Telemetry

For the past several years, we have been designing, implementing, and evaluating tools and techniques to support a telemetry-based approach to software project management as part of Project Hackystat. Figure 1 illustrates the overall architecture of the system. First, the project development environment must be 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 service. 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 telemetry stream and send email when these conditions occur.

Hackystat supports the following general classes of software project telemetry:

- *Development telemetry* is data gathered by observing the behavior of project developers and managers as reflected in their tool usage, and includes information about the files they edit, the time they spend using various tools, the changes they make to project artifacts, the sequences of tool or command invocations, and so forth. Development telemetry can be gathered by attaching sensors to editors, such as Eclipse or Emacs, to Office applications such as Word or Frontpage, to configuration management tools such as CVS, to issue management tools such as Bugzilla or Jira, and so forth.

- *Build telemetry* is data gathered by observing the results of tools invoked to compile, link, and test the system. Build telemetry can be gathered from build tools like Ant, Make, or CruiseControl, testing tools like JUnit, size and complexity tools like LOCC, and so forth.

- *Execution telemetry* is data gathered by observing the behavior of the system as it executes. This telemetry can be gathered by instrumenting the run-time environment of the system to collect data about its internal state (heap size, occurrence of exceptions, etc.) as well as by tools that perform load or stress testing of the system, such as JMeter.

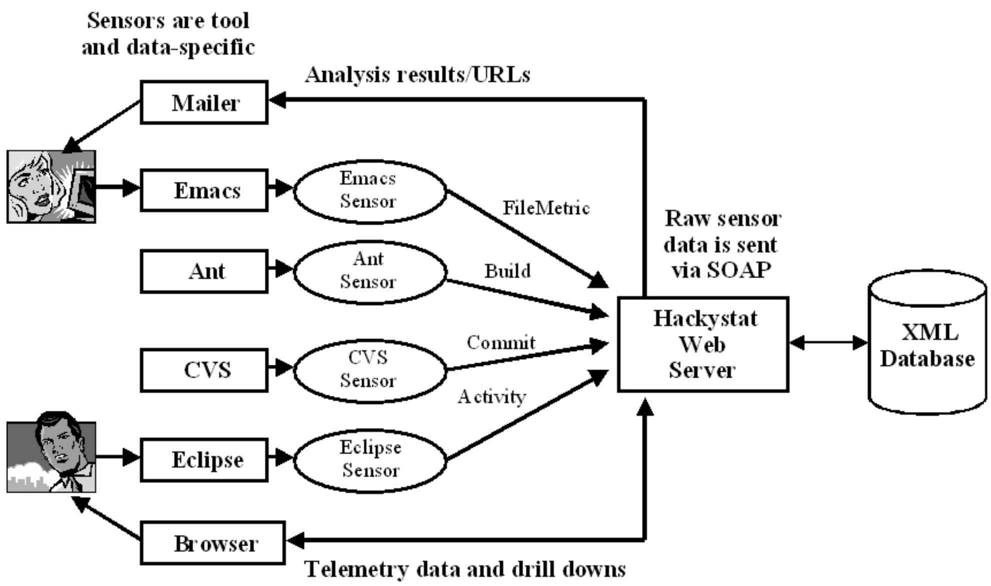

Figure 1. The basic architecture of Hackystat. Sensors are attached to tools directly invoked by developers (such as Eclipse or Emacs) as well as to tools implicitly manipulated by developers (such as CVS or an automated build process using Ant).

- *Usage telemetry* is data gathered by observing the behavior of users as they interact with the system, such as the frequency, types, and sequences of command invocations during a given period of time in a given subsystem.

For a description of the specific sensors and data types currently supported by Hackystat, see Section 7, Sidebar.

The path from sensors to the telemetry report displayed in Figure 2 involves several steps. First, "raw" sensor data is collected by observing behavior in various client systems and then sent to the Hackystat server, which it is persisted in an XML-based repository. The raw sensor data is abstracted into "DailyProjectData" instances, which can involve the synthesis of sensor data from multiple group members and/or multiple sensors into a higher level representation of sensor data for a given project and day. Sets of DailyProjectData instances are then manipulated by "Reduction Functions", which emit a sequence of numerical telemetry values for a given project and time scale of days, weeks, or months.

The previous steps occur "below the hood", as part of the software implementation of telemetry support. An important benefit of Hackystat is its explicit support for the exploratory nature of telemetry-based decision making. We have designed a simple "Telemetry Display Language" that can be used with the Hackystat web server to interactively define telemetry streams and specify how they should be composed together into charts and reports for presentation to developers and managers.

Figure 2 shows an example of end results of this process. In this scenario, project members were interested in seeing if they could detect a relationship between aggregate code churn (the lines added and deleted from the CVS repository by all members of the project) and aggregate build results (the number of build attempts and failures on a given day via invocation of the Ant build tool). To check this hypothesis, they used the web interface to define four telemetry streams (lines added, lines deleted, build attempts, and build failures) and defined a telemetry report called BuildAndChurn to display these four streams in a pair of charts. Note that Telemetry Reports are always defined without reference to a specific project or time interval. The specification of the project and time interval for presentation in the report is specified when

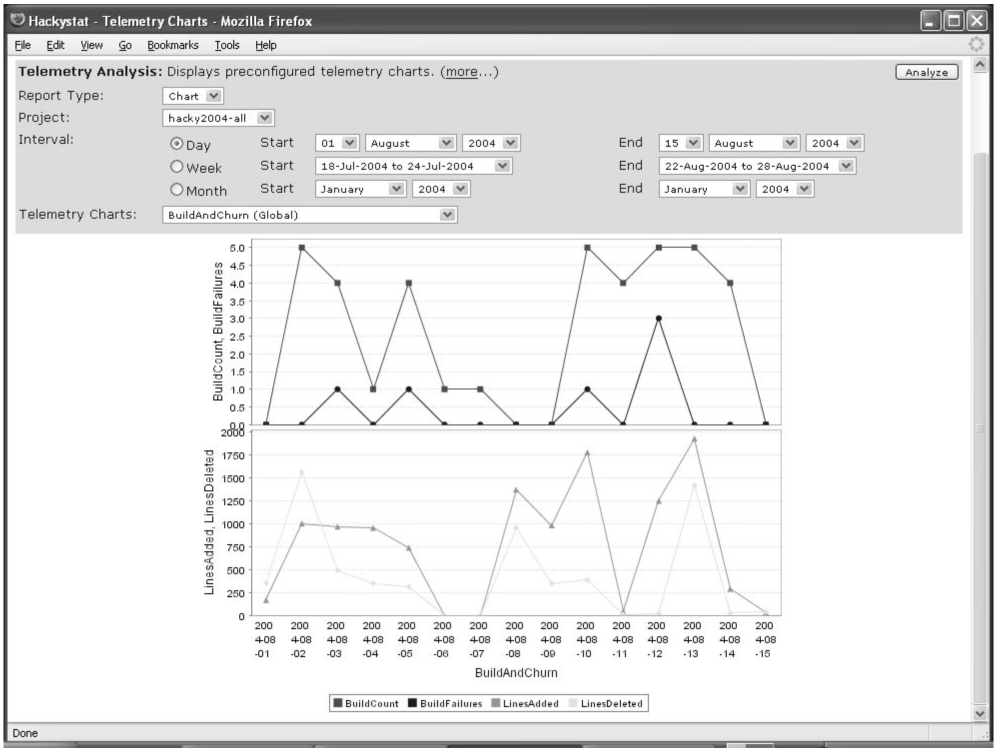

Figure 2. A telemetry report that compares code churn (lines added and lines deleted) to build results (number of build attempts and number of failures.

the report is generated, not as part of its definition. Thus, project members can now run this telemetry report over differing sets of days, or else change the time scale to Weeks for Months to see if different trends emerge from these alternative perspectives. In addition, once defined, members of other projects on this server could use the BuildAndChurn telemetry report to see if it adds decision-making value to their project management activities.

Furthermore, this aggregate report reflecting project-wide data may be only the beginning of the telemetry investigation process. Most reduction functions also support parameters that allow users to create telemetry streams restricted to data collected from a particular module in the project, or from a particular project member, or both. In the above case, for example, project members might decide based upon this Telemetry Report to define another one that presents churn data on a module-by-module basis, in order to investigate whether one module contributes inordinately to the build results.

4. The Telemetry Control Center

For a project of even moderate size and complexity, the number of possible telemetry charts and reports quickly explodes. For example, the Hackystat development project is monitored by ten different sensor data streams, across two dozen modules, from nine active developers. Given that each telemetry stream can be composed from one or more sensor data streams, one or more project modules, and one or more developers, you can see the problem: which of the literally thousands of possible charts should we be monitoring?

In our development group, we decided to address this problem by creating a new interface to the telemetry data that would enable us to passively monitor telemetry in a way that a standard web browser would not allow. We call this interface the “Telemetry Control Center”, as shown in Figure 3.

The Telemetry Control Center consists of a standard PC with a multihead video display card that is attached to nine 17” LCD panels, mounted on the wall in our

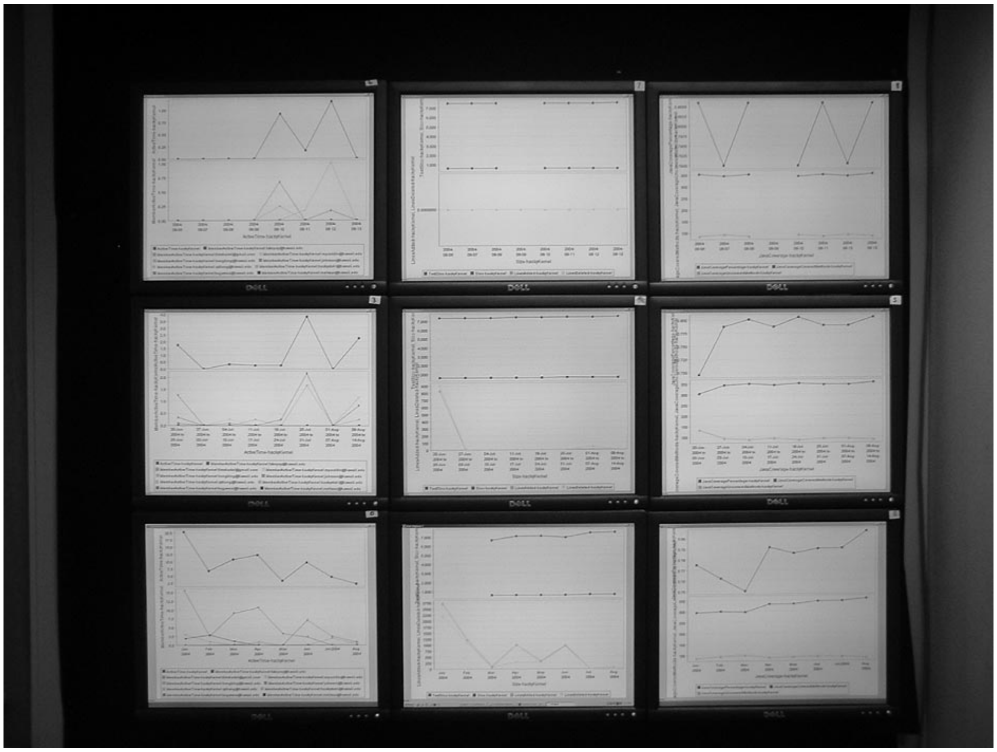

Figure 3. The Telemetry Control Center, showing one “scene” consisting of nine telemetry reports. The associated TelemetryViewer software controls the TCC by automatically cycling through a set of scenes at a predefined interval. This telemetry viewer is configured to show a dozen separate scenes, each displayed for two minutes.

laboratory. We implemented a new client-side software system called the TelemetryViewer, which periodically requests Telemetry Reports from the Hackystat server, retrieves the resulting image file, and displays them on screens. The TelemetryViewer reads in an XML configuration file at startup, which tells it which reports to retrieve, where to display them, and how long to wait before retrieving the next set of reports. The default behavior of the TelemetryViewer is to automatically and repeatedly cycle through the set of telmetry “scenes” specified in the XML configuration file.

The Telemetry Control Center frees us from the “tyranny of the browser”, by making a sequence of telemetry report sets continuously available without any action on the part of developers. It also enables us to more easily look for relationships between telemetry streams, since the system can display nine telemetry reports simultaneously. Finally, it provides a new kind of passive awareness about the state of the project to all developers; rather than having to decide to generate a report or wait for a weekly project update meeting, developers can simply glance at the Telemetry Control Center whenever they are passing though the lab to get a perspective on the state of development.

5. Lessons Learned

Our first lesson learned is that software project telemetry can provide useful support for project management decision making. For example, telemetry data has revealed to us a recent, subtle slide in testing coverage over the past six months that has co-occurred with two episodes of significant refactoring (and resulting code churn). As a result, we are allocating additional effort to software review with a focus on assessing the test quality of new modules. We hope that this project management decision will lead to a reversal of the declining coverage trend.

Hackystat provides an open source reference framework for Software Project Telemetry, but Hackystat is not the only technology available. Commercial measurement tools can also provide infrastructure support, or your organization could decide to develop technology in-house. The key issue is to preserve the essential properties of software project telemetry.

We have learned that having an automated daily build mechanism adds significant value to software project telemetry. It both provides a convenient hook into which you can add sensors to reliably obtain daily information about product measures, but also provides a kind of heart beat for the development project that makes all the metrics more comparable, accessable, and current.

As with any measurement approach, it is possible to misinterpret and misuse software project telemetry data. For example, telemetry data is intrinsically incomplete with respect to measuring “effort”. Hackystat implements a measure called Active Time, which is the time developers and managers spend editing files related to a given project in tools such as Eclipse, Word, or Excel. However, many legitimate and productive activities, including meetings, email, and hallway conversations are outside the scope of telemetry-based measurement. Telemetry cannot measure “effort” in its broadest sense, and a small value of Active Time by a project member does not necessarily imply that they are not contributing a great deal of productive effort to the project. Indeed, some organizations may decide not to collect measures such as Active Time, simply because it is susceptible to misinterpretation and/or abuse.

The adoption of a software project telemetry approach to measurement and decision making tends to exert a kind of gravitational force toward increased use of tools for managing process and products. For example, a small development team might begin by informally managing tasks and defects using email or index cards. As they start to adopt telemetry-based decision making, they will inevitably want to relate development process and product characteristics to open tasks, defect severity levels, and so forth, but not be able to do so unless they move to an issue management tool such as Bugzilla or Jira that enables sensor-based measurement.

6. Future directions

Does software project telemetry provide a silver bullet that solves all of the problems associated with metrics-based software project management and decision making? Of course not. While software project telemetry does address certain problems inherent in traditional measurement, and provide a new approach to more local, in-process decision-making, it provides its own set of issues that must be addressed by future research and practice.

First, the decision-making value of telemetry data is only as good as the quality and diversity of data that can be obtained by sensors. Clearly, there is some threshold for sensor data, beneath which the decision-making value of software project telemetry is compromised. But what is this threshold, and how does it vary with the kinds of decision-making required by the development group? What set of sensors and sensor data types are best suited to what project development contexts?

Second, what are the intrinsic limitations to telemetry-based data? A good way to investigate this question involves qualitative, ethnographic research, in which a researcher trained in these methods observes a software development group to learn what kinds of information relevant to project management decision-making occur outside of the realm of telemetry data.

Third, while manual investigation of telemetry streams and their relationship to each other is certainly an important and necessary first step, the sheer number of possible relationships and interactions means that only a small percentage of them can be inspected and monitored manually on an ongoing basis. An intriguing future direction is to explore the use of data mining and clustering algorithms to see if they can reveal relationships in the telemetry data that might not be discovered through manual exploration.

Finally, Hackystat is an open source system that is freely available for download and use. We encourage interested theorists and practitioners to visit the developer services website at http://hackydev.ics.hawaii.edu, where access to source, binaries, and documentation is available. To try out Hackystat, we also maintain a public server at http://hackystat.ics.hawaii.edu.

7. Sidebar: Sensors and Sensor Data Types

Hackystat provides an extensible architecture with respect to both “sensors”, or the software plugins associated with development tools, and “sensor data types”, which describe the structure of a given type of raw metric data. This mapping is not one-to-one: for example, the Eclipse sensor can send Activity, FileMetric, and Review sensor data types, and the FileMetric sensor data type can be collected by both IDE and Size metric sensors. The following lists describe the range of currently available sensors and sensor data types; each organization can decide whether or not to enable these facilities, or whether to implement their own custom extensions to better facilitate their own needs.

Hackystat sensors are currently implemented for the following tools:

- *Interactive development environments*, including Eclipse, Emacs, JBuilder, Vim, and Visual Studio;
- *Office productivity applications*, including Excel, Word, Powerpoint, and Frontpage;
- *Build tools*, including Ant and the Unix command line;
- *Size measurement tools*, including CCCC and LOCC;
- *Testing tools*, including JUnit and JBlanket;
- *Configuration management*, including CVS and Harvest;
- *Defect tracking tools*, including Jira;

Hackystat sensor data types include:

- *Activity*, which represents data concerning the active time spent by developers in their IDE;
- *BufferTransitions*, which represents the sequence of files visited by developers;
- *Review*, which represents data on review issues generated during code or design inspections;
- *FileMetric*, which represents size information about files;
- *Build*, which represents data about the occurrence and outcome of software builds.
- *Perf*, which represents data about the occurrence and outcome of performance analysis activities such as load testing;
- *CLI*, which represents data about the occurrence of command line invocations;
- *UnitTest*, which represents data about the occurrence and outcome of unit test invocations;
- *Coverage*, which represents data about the coverage obtained by unit testing activities;
- *Commit*, which represents data about configuration management commit events by developers;
- *Defect*, which represents information about the posting of defect reports to defect tracking tools by developers or users

8. Acknowlegements

We gratefully acknowledge support for Project Hackystat by the NSF and NASA as part of their joint program on Highly Dependable Systems, by Sun Microsystems as part of the DARPA High Productivity Computing Systems program, and by IBM as part of the Eclipse Innovation Grant program.

References

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

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] T. DeMarco. *Controlling Software Projects*. Yourdon Press, New York, 1982.

[4] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, 1997.

[5] L. Finkelstein. What is not measureable, make measurable. *Measurement and Control*, 1982.

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

[7] S. H. Kan, J. Parrish, and D. Manlove. In-process metrics for software testing. *IBM Systems Journal*, 40(1), 2001.

[8] P. Kulik and M. Haas. Software metrics best practices – 2003. Technical report, AcceleraResearch, Inc., March 2003.