IMPROVING SOFTWARE DEVELOPMENT MANAGEMENT WITH SOFTWARE PROJECT TELEMETRY

A DISSERTATION PROPOSAL SUBMITTED TO THE GRADUATE DIVISION
OF THE UNIVERSITY OF HAWAI‘I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

PH.D.

IN

COMPUTER SCIENCE

By
Qin Zhang
qzhang@hawaii.edu
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawaii

Dissertation Committee:

Philip Johnson, Chairperson
Daniel Suthers
Martha Crosby
Wes Peterson
Daniel Port

October 26, 2005
Version 2.0

Abstract

Software development is slow, expensive and error prone, often resulting in products with a large number of defects which cause serious problems in usability, reliability and performance. To combat this problem, software measurement provides a systematic and empirically-guided approach to control and improve development processes and final products. Experience has shown excellent results so long as measurement programs are conscientiously implemented and followed. However, due to the high cost associated with metrics collection and difficulties in metrics decision-making, many organizations fail to benefit from measurement programs.

In this dissertation, I propose a new measurement approach – *software project telemetry*. It addresses the "metrics collection cost problem" through highly automated measurement machinery – sensors are used to collect metrics automatically and unobtrusively. It addresses the "metrics decision-making problem" through intuitive high-level visual perspectives on software development that support in-process, empirically-guided project management and process improvement. Unlike traditional metrics approaches which are primarily based on historical project databases and focused on model-based project comparison, software project telemetry emphasizes project dynamics and in-process control. It combines both the precision of traditional project management techniques and the flexibility promoted by agile community.

The main claim of this dissertation is that software project telemetry provides an effective approach to (1) automated metrics collection, and (2) in-process, empirically-guided software development process problem detection and analysis. Three case studies will be conducted to evaluate the claim in different software development environments:

1. **[completed]** A pilot case study with student users in software engineering classes to (1) test drive the software project telemetry system in preparation for the next two full-scale case

studies, and (2) gather the students' opinions when the adoption of the technology is mandated by their instructor.

2. **[planned]** A case study in CSDL to (1) use software project telemetry to investigate and improve its build process, and (2) evaluate the technology at the same time in CSDL (an environment typical of traditional software development with close collaboration and centralized decision-making).

3. **[planned]** A case study at Ikayzo with open-source project developers (geologically-dispersed volunteer work and decentralized decision-making) to gather their opinions about software project telemetry.

The time frame of this research is as follows. The implementation of the software project telemetry system is complete and deployed. I have finished the first pilot case study. I will start both the second and third case studies from October 2005, and they will last 4 - 6 months. I wish to defend my research in May or August 2006 if everything goes according to plan.

Table of Contents

Chapter 1

Introduction

1.1 Software Crisis

Software “production” is inherently different from manufacturing production. Every software project is unique. Despite tremendous research effort invested by the software engineering community for the past several decades to build reliable software efficiently and effectively, software development methods, as currently practiced, still remain largely an art. Software development is slow, expensive and error prone, often resulting in products with large number of defects which cause serious problems in usability, reliability and performance.

According to *the Chaos Report* [78] published by the Standish group, companies in the United States spent around $250 billion on software development per year on approximately 175,000 projects. Only 16% of these projects finished on schedule and within budget. 31% were cancelled before completion mainly due to quality problems for losses of about $81 billion. 53% exceeded their original budgets by an average of 189% for losses of about $59 billion. Those projects that made it to completion delivered an average of 42% of the planned features. The direct cost of these failures and overruns were just the tip of the iceberg. The loss of indirect opportunity costs were not measured in the report, but could easily be trillions of dollars.

A report [68] from the Software Engineering Institute (SEI) indicated that out of 542 software organizations participating in a CMM maturity assessment, 67% of them were at the lowest maturity level (level 1), and 20% were at maturity level 2. The software process at level 1 was characterized as *ad hoc* and sometimes *chaotic*. Inputs to the process were ill-defined, and the transition from inputs to final products was uncontrolled. The development process was so reactive that management control was impossible. The development process at level 2 was better, because earlier successes on projects with similar applications could be repeated. However, there was still no visibility into how software products were produced, and any pertubation to development teams or development resources could easily cause project failure. Combined, 87% of the software organizations surveyed were unable to control their development processes, nor were they unable to consistently develop software products on schedule and within budget.

Uncontrollable and non-repeatable processes cause many problems to software development organizations. For example, it becomes hard to ensure software quality, hard to make reliable effort and schedule estimation, and impossible to allocate resources efficiently.

1.2 Software Measurement, Process Improvement and Predictions

It is conventional wisdom that *you can neither predict nor control what you cannot measure* [25]. Consistent measurement is a key component in establishing a scientific basis for software engineering. Software metrics are capable of quantifying software products and their development processes in an objective way. They make certain aspects of processes and products more visible, and give us better understanding of relationships between development activities and the attributes of software products they affect. As a result, various measurement programs have been developed to improve software organizations' development processes and their capability to produce software products in a controllable and repeatable manner.

Effective measurement programs help software organizations understand their capabilities, so that they can develop achievable plans for producing and delivering software products. Furthermore, continual measurement can provide an effective foundation for managing process improvement activities. The end result is that software organizations can have controllable and repeatable development processes, and thus the ability to make reliable predictions about their development activities.

Indeed, software measurement is always at the core of software process improvement and assessment programs, such as PSP [41, 42], CMM [40, 67, 41], ISO 9001 [44], SPICE [45, 29] and BOOTSTRAP [55]. Industrial experience [34] has demonstrated that so long as measurement programs are conscientiously followed, they can help software organizations achieve improved development processes, both in the short run and in the long run. Controllable and repeatable processes are essential for software organizations to make reliable predictions about their development activities, such as those in SLIM [72, 71] and COCOMO [10, 11], so that they can make realistic and achievable plans.

1.3 Problem Statement

Despite the potential of software measurement in theory and positive experiences in reality, effective application appears far from mainstream in practice. For example, a recent case study [54] surveyed 630 software professionals. Only 27% of the "best practice" organizations responded that reliance on metrics and measurements when making project-related decisions was *very* or *extremely* important, while just 2% of "all other" organizations responded the same way.

Several reasons can be hypothesized.

All measurement activities compete for resources. An important question for a software organization committing itself to measurement program is whether the benefit from improved development processes outweighs the cost. Existing measurement programs tend to be very expensive. For example, the PSP uses manual metrics collection. Every time a compilation error occurs, the developer has to stop his/her current work, and record details about the error. It is not only tedious, but also susceptible to bias, error, omission, and delay. CMM requires that in all key process areas measurements be taken to determine the status of the activities. A study by Herbsleb, et al. [37] found that it took on average 2 years per level for a software development organization to get from CMM level 1 to level 3, and that the cost ranged from $500 to $2000 per employee per year. Quantitative measurement is explicitly addressed in CMM level 4, and Humphrey himself admitted that "the greatest potential problem with the managed process (i.e. level 4) was the cost of gathering data, and that there were an enormous number of potentially valuable measures of the software process, but such data were expensive to gather and to maintain" [39]. Due to high cost associated with metrics collection and analysis, it is a daunting task to apply measurement best practices to improve an software organization's development process in practice.

Even if we successfully overcome the issues with metrics collection, there is still difficulty of using software measures for the purpose of project management and process improvement. Traditional approaches use "historical project database" as a baseline for comparison with metrics from the current project, which tend to yield results not easily translated into practice. They typically involve the following basic procedure: (1) collect a set of process and product metrics, such as size, effort, complexity, and defects, for a set of completed software projects, (2) generate a model to fit the observed data, (3) and claim that the model can be used to predict characteristics of future projects. Project management and process improvement is based on the predictions made by the models. For example, a model might predict that a future project of size S would require E person-months of effort; another model might predict that a future implementation of a module with complexity C would be prone to defects with density D.

This model-based process prediction technique is used in many forms, such as in the PSP and COCOMO. It faces a number of limitations. First, the predictive power of these models depends crucially on how well model calibration is performed. In order to use the model off-the-shelf, practitioners must confirm that the set of projects used to calibrate the model are “comparable” to the project they wish to predict. Otherwise, the model must be recalibrated using the data in the organization’s historical project database, to avoid the problem of comparing apples to oranges. This involves replicating the model-building method within the practitioner’s organization, with the risk that the applications, personnel, and resources may have already changed, and the context of future projects may differ from those in the historical project database. Second, model-based process prediction assumes the process of a software organization is predictable. However, according to the SEI survey [68] mentioned in Section 1.1, 67% of the surveyed organizations were at the lowest CMM maturity level. By definition, the software processes at that level are *ad hoc* and sometimes *chaotic*, and they change as work changes. As a result, it is generally impossible to make predictions for these organizations. Lastly, and most fundamentally, the majority of the models that are available are built to compare a set of finished projects. This may be useful in initial project planning, but we are more interested in managing a project and guiding it to successful completion. How do we compare metrics from completed projects to the one that is still in progress?

The result is the dilemma we are facing today. Almost all software organizations are aware of the benefits of a mature development process and the value of measurement programs in achieving it. However, few of them are capable of implementing a successful measurement program in practice. The difficulties lie in (1) the overhead and the cost associated with metrics collection and analysis, and (2) the difficulty in translating metrics research results into practice.

1.4 Proposed Solution - Software Project Telemetry

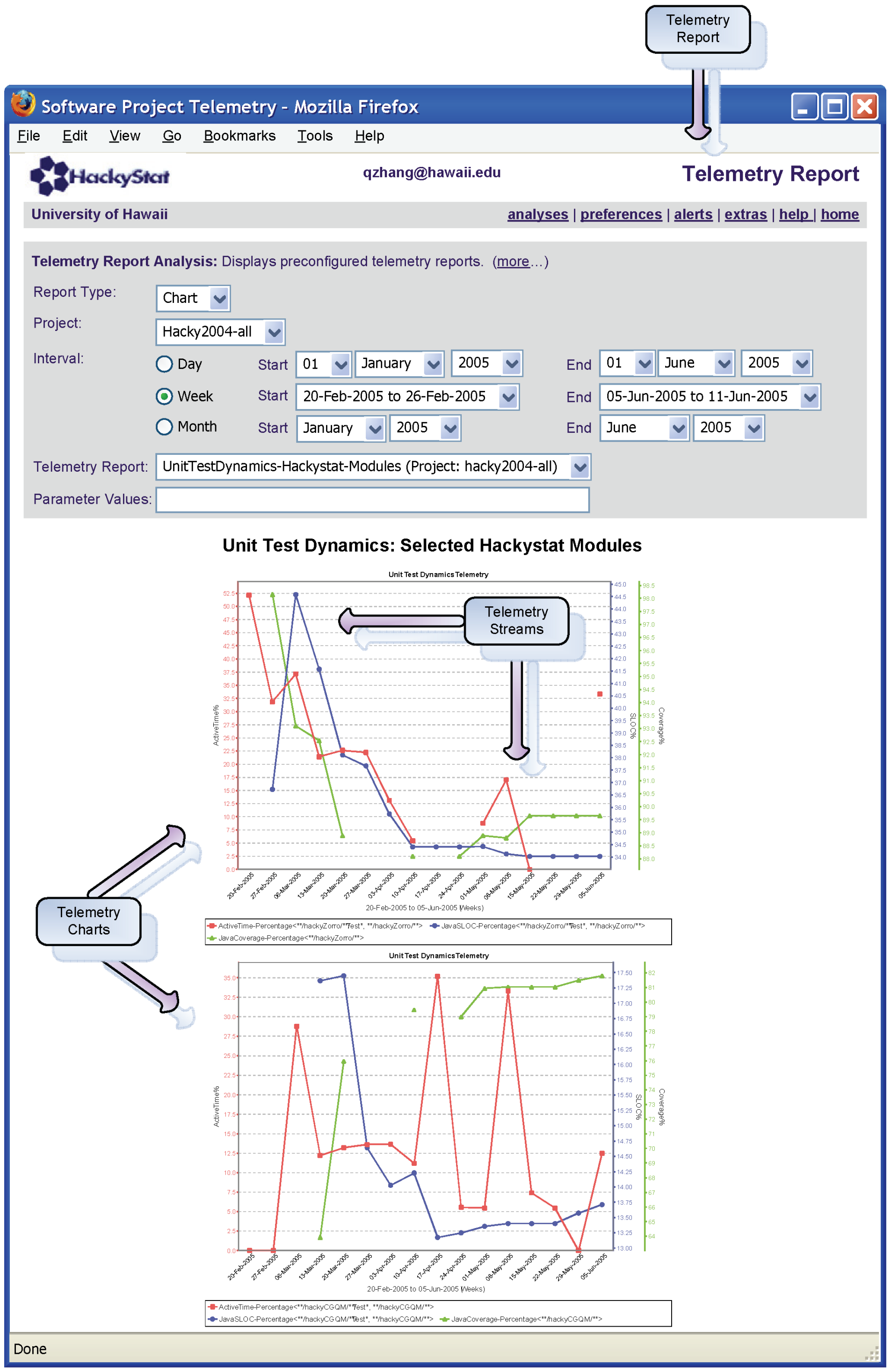
To address the problem, I propose a novel, light-weight measurement approach called *software project telemetry*. It includes both (1) highly automated measurement machinery for metrics collection and analysis, and (2) a methodology for in-process, empirically-guided software development process problem detection and diagnosis.

In this approach, sensors collect software metrics automatically and unobtrusively. Metrics are abstracted in real time to telemetry streams, charts, and reports, which represent high-level perspectives on software development. Telemetry trends are the basis for decision-making in project management and process improvement. Unlike traditional approaches which are primarily based on historical project databases and focused on comparison of similar projects, software project telemetry emphasizes project dynamics and in-process control. It combines both the precision of traditional project management techniques and the flexibility promoted by agile community.

1.4.1 Measurement Machinery in Software Project Telemetry

In software project telemetry, sensors collect metrics *automatically*. Sensors are software tools that monitor some form of state in project development environment *unobtrusively*. Some examples are (1) a sensor for an IDE monitoring developer activities, such as code editing effort, compilation attempts and results, etc., or (2) a sensor for a version control system monitoring code check-in and check-out activities, and computing *diff* information between different revisions. There can be many possibilities. However, the key is that sensors are designed to collect metrics *automatically* and *unobtrusively* in order to keep metrics collection overhead sufficiently low, so that developers are not distracted from their primary task – developing software products instead of recording process and product metrics.

The metrics collected by sensors are time-stamped, and the time stamp is always significant in metrics analysis. *Telemetry streams*, *charts*, and *reports* capture high-level perspectives on software development, while *the telemetry language* facilitates interactive exploration of these perspectives. Figure 1.1 shows the relationship between telemetry streams, charts, and reports.


Telemetry Report
Telemetry Streams
Telemetry Charts
Software Project Telemetry - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
qzhang@hawaii.edu
Telemetry Report
University of Hawaii
analyses | preferences | alerts | extras | help | home
Unit Test Dynamics Telemetry
ActiveTime%
SLOC%
Coverage%
20-Feb-2005 to 05-Jun-2005 (Weeks)
ActiveTime-Percentage<**/hackyZorro/**Test*, **/hackyZorro/**>
JavaSLOC-Percentage<**/hackyZorro/**Test*, **/hackyZorro/**>
JavaCoverage-Percentage<**/hackyZorro/**>
ActiveTime-Percentage<**/hackyCGQM/**Test*, **/hackyCGQM/**>
JavaSLOC-Percentage<**/hackyCGQM/**Test*, **/hackyCGQM/**>
JavaCoverage-Percentage<**/hackyCGQM/**>
Done

Telemetry Report Analysis: Displays preconfigured telemetry reports. (more…)

Report Type: Chart

Project: Hacky2004-all

Interval:
- Day — Start 01 January 2005 — End 01 June 2005
- [✓ Week] — Start 20-Feb-2005 to 26-Feb-2005 — End 05-Jun-2005 to 11-Jun-2005
- Month — Start January 2005 — End June 2005

Telemetry Report: UnitTestDynamics-Hackystat-Modules (Project: hacky2004-all)

Parameter Values:

Unit Test Dynamics: Selected Hackystat Modules

Figure 1.1. The Relationship between Telemetry Streams, Charts, and Reports

Telemetry Reports

A telemetry report is a named set of telemetry charts that can be generated for a specified project over a specified time interval. The goal of a telemetry report is to discover how the trajectory of different process and product metrics might influence each other over time, and whether these influences change depending upon context.

For example, Figure 1.1 shows a telemetry report consisting of two charts. Both charts show *Unit Test Dynamics Telemetry*, which is an analysis of trends in the percentage of active time[1] allocated to testing, the percentage of source code devoted to testing, and the percentage of test coverage that results from this effort and code. The charts share the same time interval and project. The only difference is that they show unit test dynamics data for two different modules in the same project. Interestingly, the unit test dynamics telemetry trends for the two modules have a very different shape, indicating differences in the underlying approach to development of these two modules.

Telemetry Charts

A telemetry chart is a named set of telemetry streams that can be generated for a specified project over a specified time interval. The goal of a telemetry chart is to display the trajectory over time of one or more process or product metrics.

For example, Figure 1.1 shows two instances of the same telemetry chart. The chart, entitled *Unit Test Dynamics Telemetry*, contains three telemetry streams: *ActiveTime-Percentage*, *JavaSLOC-Percentage*, and *JavaCoverage-Percentage*. You can see references to these three streams in the legend accompanying each chart. The legends also illustrate that telemetry streams can be parameterized: the top chart contains streams parameterized for the *hackyZorro* module, while the bottom chart contains streams parameterized for the *hackyCGQM* module.

Telemetry Streams

Telemetry streams are sequences of a single type of process or product data for a single project over a specified time interval. Telemetry streams are best thought of as a kind of abstract data type

[1] Active time is a proxy for developer effort writing and editing code inside an IDE.

representing one or more series of metric data values of a similar type. In addition, they support basic arithmetic operations.

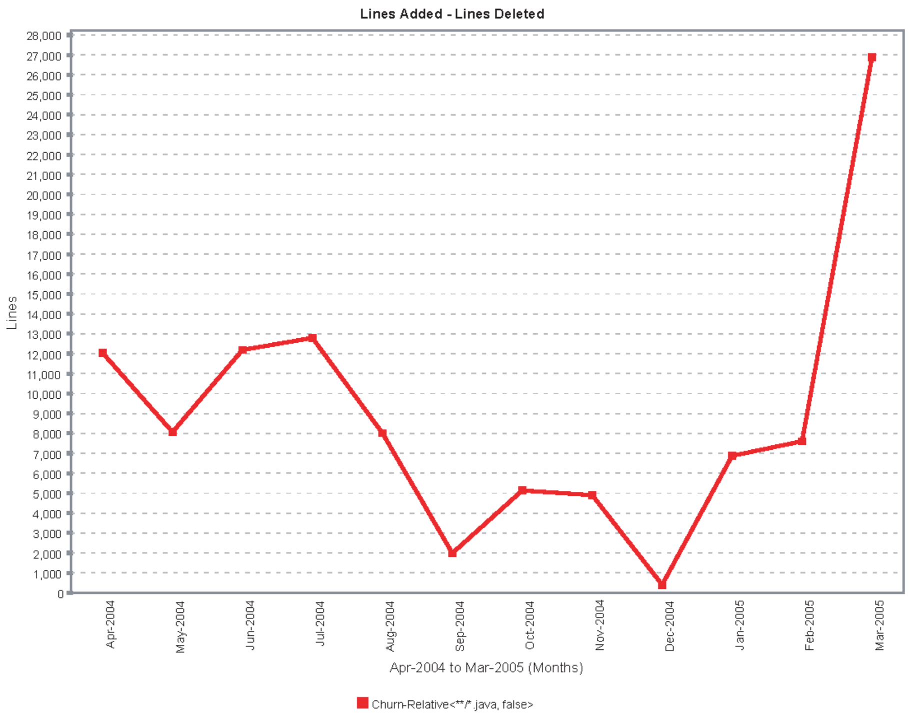
For example, Figure 1.1 shows three kinds of telemetry streams, each with its own color. The red line in each chart is an *ActiveTime-Percentage* stream, the blue line in each chart is a *JavaSLOC-Percentage* stream, and the green line in each chart is a *JavaCoverage-Percentage* stream.

The time interval covered by telemetry stream is divided into periods. The data points in each telemetry stream reflect the state of some key aspect of the software system under study during each period. The period can be relatively fine-grained such as daily, or more coarse-grained such as weekly or monthly. Two types of information are typically represented by the data points:

- Aggregation information - The metrics metrics values can be accumulated over the time period. Some examples are total coding effort, total lines added or deleted in the source code, total number of new bugs reported, etc.
- Snapshot information - The metrics are only meaningful at a specific point in time. Some examples are the size of the source code, the number of open bugs, etc. Usually, the snapshot is taken at the beginning or at the end of the period.

Figure 1.2 and 1.3 show four telemetry streams. The data points in the upper telemetry streams in the two figures represent aggregation information. In this case, they are the net increase in the number of lines in the source code. The time period in Figure 1.2 is month, and each data point represents code size increase for the entire month. The time period in Figure 1.3 is day, and each data point represents code size increase for one single day. The data points in the lower telemetry streams in the two figures represent snapshot information. In this case, they are the number of Java classes in a software system. Though the two streams use monthly and daily period respectively, the data points actually represent the snapshot of the system size at the end of each period.

The advantage of using telemetry streams is that they show the entire history of the status of some state in project development environment, and thus help project manager detect changes in the development process. Telemetry streams may contain missing data, such as in the case of the lower one in Figure 1.3. While complete data provide the best support for project management, occasional drop of data point should have little impact on their value for decision-making. As a result, analyses built on top of telemetry streams can exhibit graceful degradation, providing value even when only partial data is available.



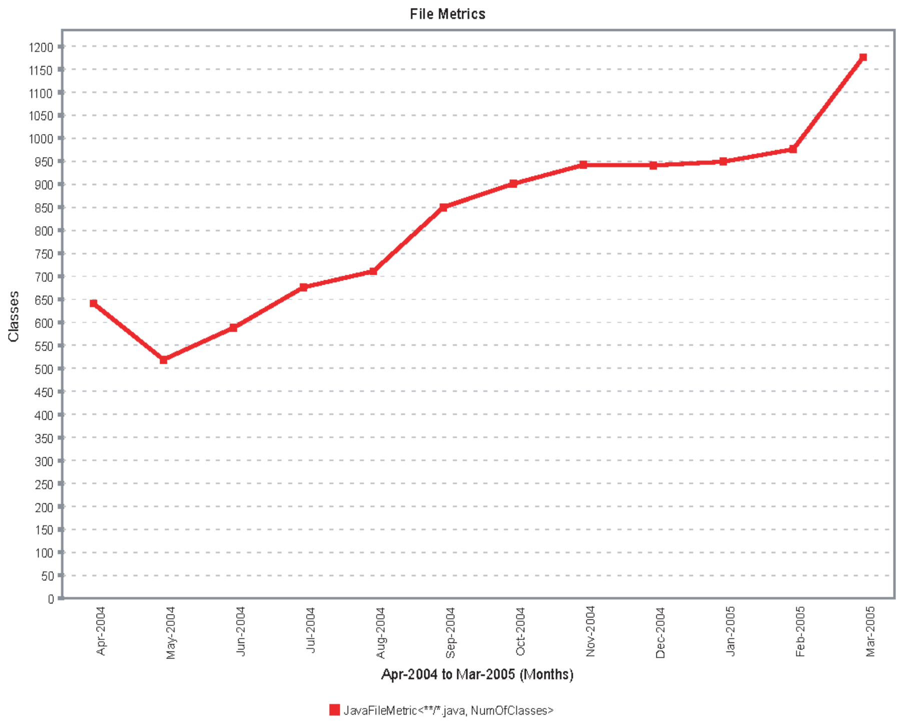
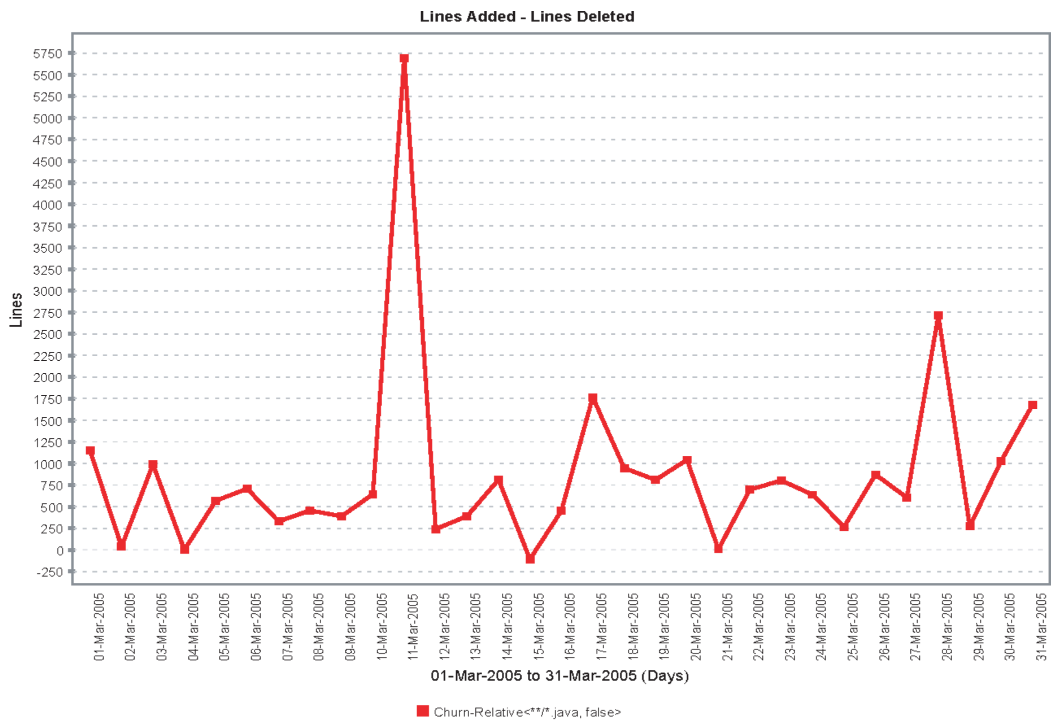


Figure 1.2. Telemetry Streams (Monthly)



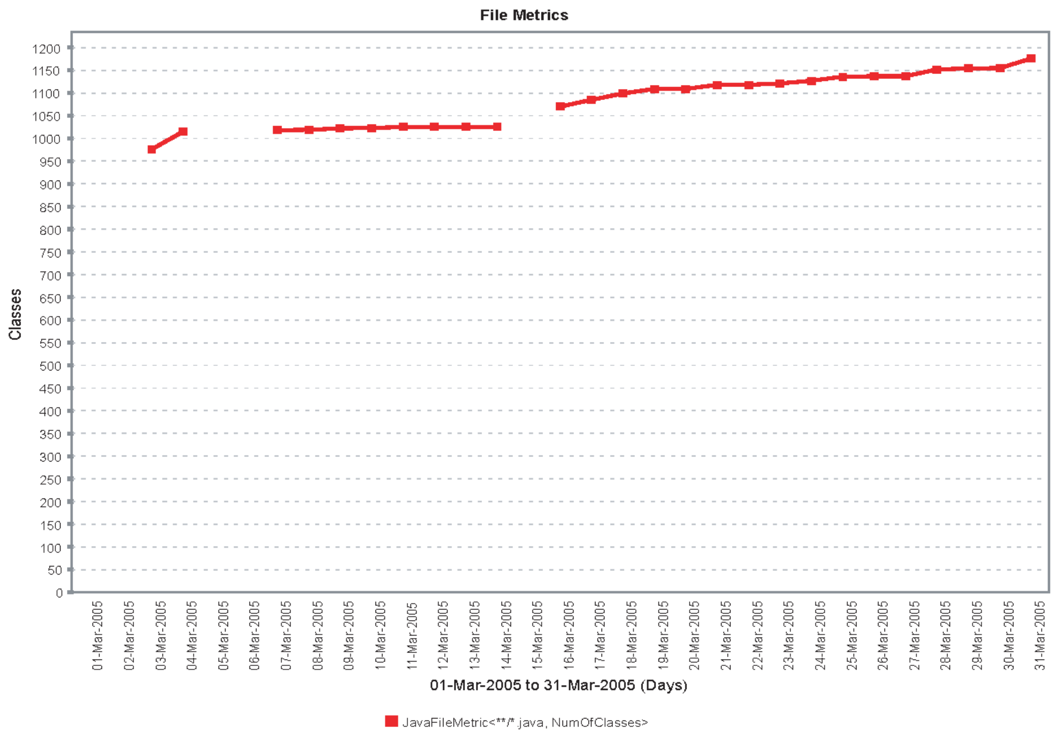


Figure 1.3. Telemetry Streams (Daily)

Telemetry Language

Telemetry language provides a flexible mechanism to construct telemetry streams, charts, and reports, and facilitates interactive exploration of different perspectives on software development. The language has the following simple syntax, and the details can be found in Section 2.4.

```
streams <StreamName> (<ParameterList>) = {
  <DocumentationString>, <UnitLabel>,
  <ReductionFunctionExpresion>
};

chart  <ChartName>  (<ParameterList>) = {
  <ChartTitile>, <StreamExpression>
};

report <ReportName> (<ParameterLilst>) = {
  <ReportTitle>, <ChartExpression>
};
```

1.4.2 Process Methodology in Software Project Telemetry

Telemetry streams contain software process and product metrics, and they are the basis of project management and process improvement. Telemetry charts and reports provide visualizations for project managers that help monitor software development status, and that can detect undesirable process changes. By juxtaposing different telemetry streams for the same time period, a project manager can detect co-variances between them. For example, one might find that a drop in test coverage is always associated with an increase in the number of open bugs. Depending on what interval scale is used, one may even find that a decrease in test coverage is always followed by an increase in open bugs[2]. This kind of information is important to software project management, because it suggests a plausible causal relationship – low test coverage causes more bugs to slip through to the production stage. Based on this information, the project manager can implement changes to increase test coverage, and continue to use telemetry streams to monitor whether it indeed results in a decrease of the number of open bugs. At the same time, the project manager can monitor other telemetry streams, and check whether this corrective action has any unintended side

[2]The observation of timing sequence of events is not always guaranteed. Fine-grained intervals may introduce noise, while coarse-grained intervals may lose information. This is one of the reasons why telemetry analyses offer different interval scales, such as daily, weekly, and monthly.

effect to the development process. For example, he/she may wish to monitor productivity related telemetry streams to make sure that there is no reduction in developer productivity.

In general, telemetry-based software project management involves cycles of:

1. *Problem Detection* — Use telemetry streams to monitor the development of a software project. Detect anomalies and undesirable trends in telemetry streams.
2. *Process Improvement Hypothesis Generation* — Determine possible causes for the problem, and possible measures to correct it.
3. *Process Change Implementation* — Implement corrective measures.
4. *Hypothesis Validation and Impact Analysis* — Determine whether the problem goes away after corrective measures are implemented, and whether there are any unintended side effects caused by the corrective measures.

The cycle continues until the project reaches completion.

1.5 Thesis Statement

The main claim of this thesis is that software project telemetry provides an effective automated approach to in-process, empirically-guided software development process problem detection and diagnosis.

Compared to traditional model-base process prediction approaches, management with software project telemetry has many advantages.

Software project telemetry is easier to use and cheaper to implement. It does not require software organizations to accumulate process and product metrics of finished projects in a historical database. Nor does it require expensive and error-prone model calibration before it can be used to make predictions. Instead, it focuses on evolutionary processes in development, and relies on metrics from an earlier stage of product development of the same project to make short-term predictions. For example, if system test coverage used to be almost 100% but has been gradually dropping over time, then it may be a signal for management to re-allocate resources to improve project quality assurance. As a result, software project telemetry is best suited for in-process monitoring and control.

Software project telemetry is robust. The information contained in telemetry streams is seldom affected when there is occasional metrics drop out, and analyses still provide decision-making value even if metrics collection starts midway through a project.

Software project telemetry is flexible. There are no required set of metrics to be collected. Different software organizations can collect different set of metrics according to their objectives, cost-benefit trade-offs, and measurement capabilities. For example, organizations with low process visibility can start with simple metrics such as source code size, and more metrics can be collected as their process matures and visibility increases.

1.6 Evaluation

My dissertation claim will be evaluated through three case studies:

- A case study with student users in software engineering classes to gather their opinions about software project telemetry. **[completed]**
- A case study in CSDL using software project telemetry to investigate and improve the build process of a large-scale software development project.
- A case study at Ikayzo with open-source project developers to gather their opinions about software project telemetry.

1.7 Anticipated Contributions

The anticipated contributions of this research will be:

- The concept of software project telemetry as an effective automated approach to in-process, empirically-guided software development process problem detection and diagnosis.

- An implementation of software project telemetry which allows continuous monitoring of software project status, as well as generating and validating software process improvement hypothesis.

- The insights gained from the case studies regarding how to use software project telemetry effectively for project management and process improvement, as well as the adoption barrier of the technology.

1.8 Dissertation Proposal Organization

This dissertation proposal is organized into the following chapters:

- Chapter 1 is this chapter where the problem statement and proposed solution is described.
- Chapter 2 describes software project telemetry in detail and how it facilitates software project management and process improvement.
- Chapter 3 relates the current research to the broader context of existing work.
- Chapter 4 details the design and implementation of software project telemetry.
- Chapter 5 gives a brief review of research methods and discusses the evaluation strategies of software project telemetry.
- Chapter 6 reports on a completed pilot case study with student users in software engineering classes in which their opinions about software project telemetry were gathered.
- Chapter 7 outlines a planned case study in CSDL using software project telemetry to investigate and improve the build process of a large-scale software development project.
- Chapter 8 outlines a planned case study at Ikayzo with open-source project developers gathering their opinions about software project telemetry.
- Chapter 9 concludes this dissertation proposal with a discussion of the anticipated contributions and future directions.

Chapter 2

Software Project Telemetry

Software project telemetry is a novel, light-weight, and extensible software measurement approach proposed in this thesis. It includes both (1) highly automated measurement machinery for metrics collection and analysis, and (2) a methodology for in-process, empirically-guided software development process problem detection and diagnosis. In this approach, sensors collect software metrics automatically and unobtrusively. Metrics are abstracted in real time to telemetry streams, charts, and reports, which represent high-level perspectives on software development. Telemetry trends are the basis for decision-making in project management and process improvement. Unlike traditional approaches which are primarily based on historical project database and focused on comparison of similar projects, software project telemetry focuses on project dynamics and in-process control. It combines both the precision of traditional project management techniques and the flexibility promoted by agile community.

This chapter is organized into the following sections. Section 2.1 gives an overview of software project telemetry and its essential characteristics. Section 2.2 discusses sensor-based metrics collection. Section 2.3 discusses telemetry streams, charts, and reports, which capture high-level perspectives on software development. Section 2.4 discusses the telemetry language, which provides a flexible mechanism to construct telemetry streams, charts, and reports, and which facilitates interactive exploration of different perspectives on software development. Section 2.5 discusses the telemetry-based methodology for project management and process improvement. Section 2.6 summarizes this chapter.

2.1 Overview

Encyclopedia Britannica defines telemetry as *"highly automated communication process by which data are collected from instruments located at remote or inaccessible points and transmitted to receiving equipment for measurement, monitoring, display, and recording"*. Perhaps the highest profile user of telemetry is NASA, where telemetry has been used since 1965 to monitor space flights staring from the early Gemini missions to the modern Mars rover flights. Telemetry data, collected by sensors attached to a space vehicle and its occupants, are used for many purposes, such as gaining better insight into mission status, detecting early signals of anomalies, and analyzing impacts of mission adjustments.

Applying this concept to software project management, I define *software project telemetry* as an automated software process and product measurement approach. In this approach, sensors unobtrusively collect time-stamped software metrics, which are abstracted into telemetry streams in real time. Telemetry streams are the basis for project management and process improvement. Telemetry charts and reports provide visualization. By detecting changes and covariance among different telemetry streams, software project telemetry enables a more incremental, visible, and experiential approach to project decision-making. It has both the precision of traditional project management techniques and the flexibility promoted by agile community.

Software project telemetry has following essential characteristics:

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, software developers are working in a "remote or inaccessible location" from the perspective of metrics collection activities. This contrasts with software metrics data that require human intervention or developer effort to collect, such as PSP/TSP metrics [41].

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 [10, 11], where the time at which the calibration data was collected about the project is not significant.

3. *Software project telemetry data is continuously updated 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 provide best support for project management, the analyses are not brittle: they still provide value even if complete sensor data are not available. For example, telemetry analyses can still provide decision-making value even if data collection starts 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 various time scales. 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.

2.2 Sensor-based Data Collection

In software project telemetry, software metrics are collected *automatically* by sensors that *unobtrusively* monitor some form of state in the project development environment. Sensors are pieces of software, and they collect both *process* and *product* metrics.

Sensors collecting process metrics are typically implemented in the form of plug-ins, which are attached to software development tools in order to continuously monitor and record their activities in the background. Some examples are listed below:

- A plug-in for an IDE (integrated development environment) such as Visual Studio [80], and Eclipse [28]. It can record individual developer activities automatically and transparently, such as code editing effort, compilation attempts and results, etc.

- A plug-in for a version control system, such as Rational Clear Case [17], and CVS [23]. It can monitor code check-in and check-out activities, and compute *diff* information between different revisions.

- A plug-in for a bug tracking or issue management system, such as Bugzilla [13], and Jira [47]. Whenever an issue is reported or its status has changed, the sensor can detect such activities and record the relevant information.

- A plug-in for an automated build system, such as Cruise Control [19]. It can capture information related to build attempts and build results.

Sensors collecting product metrics are typically implemented as automated analyzers for software artifacts. Typically, these analyzers need to be scheduled to run periodically in order to acquire the continual flow of metrics required by telemetry streams. To automate these tasks, one can use a *Cron* job[1], or run them as a task in some automated build system. Some examples are listed below:

- An analyzer parsing program source code to compute size or complexity information.

- An analyzer to parse the output of coverage tools, such as Clover [18], and JBlanket [46], to retrieve testing coverage information.

There are many other possibilities. One can even imagine writing an exotic sensor that retrieves project cost information from company accounting database, if such information is useful and the practice is allowed by company policy. No matter what the sensor does and regardless of the implementation details, a sensor-based approach collects metrics unobtrusively in order to keep data collection overhead low, so that developers are not distracted from their primary tasks – developing software products instead of capturing process and product metrics.

There is no chronic overhead in sensor-based metrics collection. Though setting up sensors requires effort, once they are installed and configured, sensor data collection is automatic and unobtrusive. This contrasts with traditional data collection techniques, such as paper-and-pencil based approach used in PSP/TSP [41], or tool-supported approach used in LEAP [62], PSP Studio [36], and Software Process Dashboard [82], which require either human intervention or developer effort. Even in the case of tool-supported approach, the developer still bears the chronic overhead of constantly switching back and forth between doing work and telling the tool what work is being done [49, 51].

[1] *Cron* is a *Unix/Linux* program that enables users to execute commands or scripts automatically at a specified time or date. The *Windows* equivalent is called *Scheduled Tasks*.

The fact that chronic overhead is removed from sensor-based metrics collection not only reduces the adoption barrier of the technology, but also makes it feasible for software organizations to apply measurements to a wide range of development activities and products in order to get a comprehensive quantitative view of development processes.

The sensor-based approach does impose some restrictions:

- A sensor must be developed for each type of tool we wish to monitor. This is a one-time cost. Once the sensor is developed, it can be used by different software development organizations for different projects. The Collaborative Software Development Lab has already developed a repository of over 20 sensors for commonly-used tools.

- Some metrics may not be amenable to automated data collection. An example would be developer effort. While it is possible to instrument an IDE to automatically get information such as how many hours a developer has spent on writing code, it is almost impossible to construct a sensor that knows how much total effort a developer has contributed to a project. For instance, two developers might be discussing the design of a system in the hallway. It is very difficult to collect this type of effort automatically. It is still open research question whether all important metrics can be captured by sensors or not. This thesis takes a more pragmatic view, and is only concerned with whether sensors can collect sufficient metrics so that software project telemetry can deliver value to project management and process improvement.

2.3 Telemetry Reducer, Stream, Chart, and Report

A *telemetry report* is a named set of telemetry charts that can be generated for a specified project over a specified time interval. The goal of a telemetry report is to discover how the trajectory of different process and product metrics might influence each other over time, and whether these influences change depending upon context. A *telemetry chart* is a named set of telemetry streams. The goal of a telemetry chart is to display the trajectory over time of one or more process or product metrics. A *telemetry stream* is a sequence of a single type of process or product data. Telemetry streams are best thought of as a kind of abstract data type representing one or more series of metric data values of a similar type. In addition, they support basic arithmetic operations. The details are discussed in Section 1.4.1.

The data collected by sensors are time-stamped, and this time stamp is always significant in telemetry style metrics analysis. There may not be one-to-one correspondence between sensor data and the data points in the telemetry stream. Sensor data usually represents very fine-grained low level software process or product details, while the data points in the telemetry stream represent higher level perspectives on the software system. Typically, sensor data need to be filtered, combined, aggregated, or post-processed to derive a telemetry data point. This is the job of *telemetry reducers*, which take sensor data as input, and output telemetry data points.

A *telemetry reducer* serves as the bridge between *sensor data* and *telemetry stream*. For example, suppose we want to construct a telemetry stream representing the number of open bugs for a development project on a monthly interval for the entire year 2004. Suppose we are using Bugzilla [13], and a Bugzilla sensor has been installed since the start of the project. Suppose whenever a bug is opened or closed, the Bugzilla sensor records information such as event time stamp, event type (bug open or bug close), bug id, severity, etc. In order to compute the number of open bugs for each month in 2004, the telemetry reducer needs to scan the entire bug event sensor data.

Telemetry reducer is the building block of telemetry stream. A reducer must be available in order to construct a "simple" telemetry stream[2]. Some example of general classes of software project telemetry are:

- *Development Telemetry* — These are telemetry streams generated from data gathered by observing the behavior of software developers as reflected in their tool usage, such as the information about the files they edit, the time they spend using various tools, and the changes they make to project artifacts, the sequences of tool or command invocations, and so forth. Such metrics can be collected by attaching sensors to Integrated Development Environments (e.g. Visual Studio, Eclipse, Emacs.), configuration management system (e.g. CVS [23], Clear Case [17].), issue management systems(e.g. Bugzilla [13], Jira [47].), etc.

- *Build Telemetry* — These are telemetry streams generated from data gathered by observing the results of tools invoked to compile, link, and test the system. Such metrics can be collected by attaching sensors to build tools (e.g. Make, Ant [3], Cruise Control [19].), testing tools(e.g. JUnit [52].), size and complexity counters(e.g. LOCC [58].), etc.

[2]Telemetry streams can also be generated by applying mathematical operations to existing telemetry streams. These are called "compound" telemetry streams as opposed to "simple" telemetry streams. See Section 2.4 for details.

- *Execution Telemetry* — These are telemetry streams generated from data gathered by observing the behavior of the system as it executes. Such metrics can be collected by sensors attached to the system runtime environment to gather its internal state data (e.g. heap size, occurrence of exceptions.), or to load testing tools (e.g. JMeter [48]) of the system to gather system performance data.
- *Usage Telemetry* — These are telemetry streams generated from 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 system.

2.4 Telemetry Language

Many interesting issues in software project management involve understanding the relationship between different measures. For example, we might be interested in seeing if an increased investment in code review pays off with less unit test failures, and/or increased coverage, and/or less defects reported against the reviewed modules. Such a question requires comparing a set of metrics values over time.

The *telemetry language* is a mechanism that facilitates interactive exploration of the relationships between metrics, so that developers and managers can "play with the data" to see what perspectives provide insight to their particular situation. The telemetry language supports three primitive types: *streams*, *chart*, and *report*. It allows the user to define (1) telemetry streams from sensor data, (2) telemetry charts from telemetry streams, and (3) telemetry reports from telemetry charts.

The building blocks of *streams* object are telemetry reducers, which are responsible for synthesizing metrics collected by sensors. Each reducer returns a *streams* object, which contains one or more telemetry streams. *Streams* objects can participate in mathematical operations. For example, two compatible *streams* object can be added, and the result is a new *streams* object. Each *chart* object defines rules grouping *streams* objects, while each *report* object specifies rules grouping *chart* objects.

The language has the following syntax:

```
streams <StreamName> (<ParameterList>) = {
  <DocumentationString>, <UnitLabel>,
  <ReductionFunctionExpresion>
};

chart  <ChartName>  (<ParameterList>) = {
  <ChartTitile>, <StreamExpression>
};

report <ReportName> (<ParameterLilst>) = {
  <ReportTitle>, <ChartExpression>
};
```

<ParameterList> is a possibly empty list of comma-separated identifiers defining the parameters for the *streams* instance. Parameters allow the calling entity to pass in a value to be provided to the reduction function(s) referenced inside the *streams* definition.

<ReductionFunctionExpression> comes in two flavors: a "simple" one which consists of a single invocation of a reduction function, or a "compound" one which is an expression whose operators are '+', '-', '*', and '/', and whose operands are reduction function invocations.

The complete specification of software project telemetry language can be found in Appendix A, while detailed instructions on how to use the language can be found in Appendix B.

The following telemetry language can be used to generate the telemetry report in Figure 1.1 in the previous chapter is illustrated below:

```
streams ActiveTime-Percentage(filePattern1, filePattern2) = {
  "Active Time Percentage", "ActiveTime%",

  ActiveTime(filePattern1, "false")
  / ActiveTime(filePattern2, "false")
  * 100
};

streams JavaCoverage-Percentage(filePattern) = {
  "Java Coverage Percentage", "Coverage%",

  JavaCoverage("NumOfCoveredMethods", filePattern)
  / (JavaCoverage("NumOfCoveredMethods", filePattern)
     + JavaCoverage("NumOfUnCoveredMethods", filePattern))
  * 100
};
```

```
streams JavaSLOC-Percentage(filePattern1, filePattern2) = {
  "Java SLOC Percentage", "SLOC%",

  JavaFileMetric("Sloc", filePattern1)
  / JavaFileMetric("Sloc", filePattern2)
  * 100
};

chart UnitTestDynamics-Chart(filePattern, testFilePattern) = {
  "Unit Test Dynamics Telemetry",

  ActiveTime-Percentage(testFilePattern, filePattern),
  JavaCoverage-Percentage(filePattern),
  JavaSLOC-Percentage(testFilePattern, filePattern)
};


report UnitTestDynamics-Hackystat-Report() = {
  "Unit Test Dynamics: Selected Hackystat Modules",

  UnitTestDynamics-Chart("**/hackyZorro/**",
                         "**/hackyZorro/**/Test*"),
  UnitTestDynamics-Chart("**/hackyCGQM/**",
                         "**/hackyCGQM/**/Test*")
};

draw UnitTestDynamics-Hackystat-Report();
```

2.5 Telemetry-based Project Management and Process Improvement

Basic steps of telemetry-based process improvement are illustrated in Figure 2.1. It involves cycles of problem detection, process improvement hypothesis generation, process change implementation, hypothesis validation, and impact analysis.

Following [38], a software organization is recommended to collect a basic set of metrics, such as code size, test coverage, and build results, for every project at all time. This basic set of metrics can generate a basic set of telemetry streams which provide insights into the current software products, as well as a base line for the current development process.

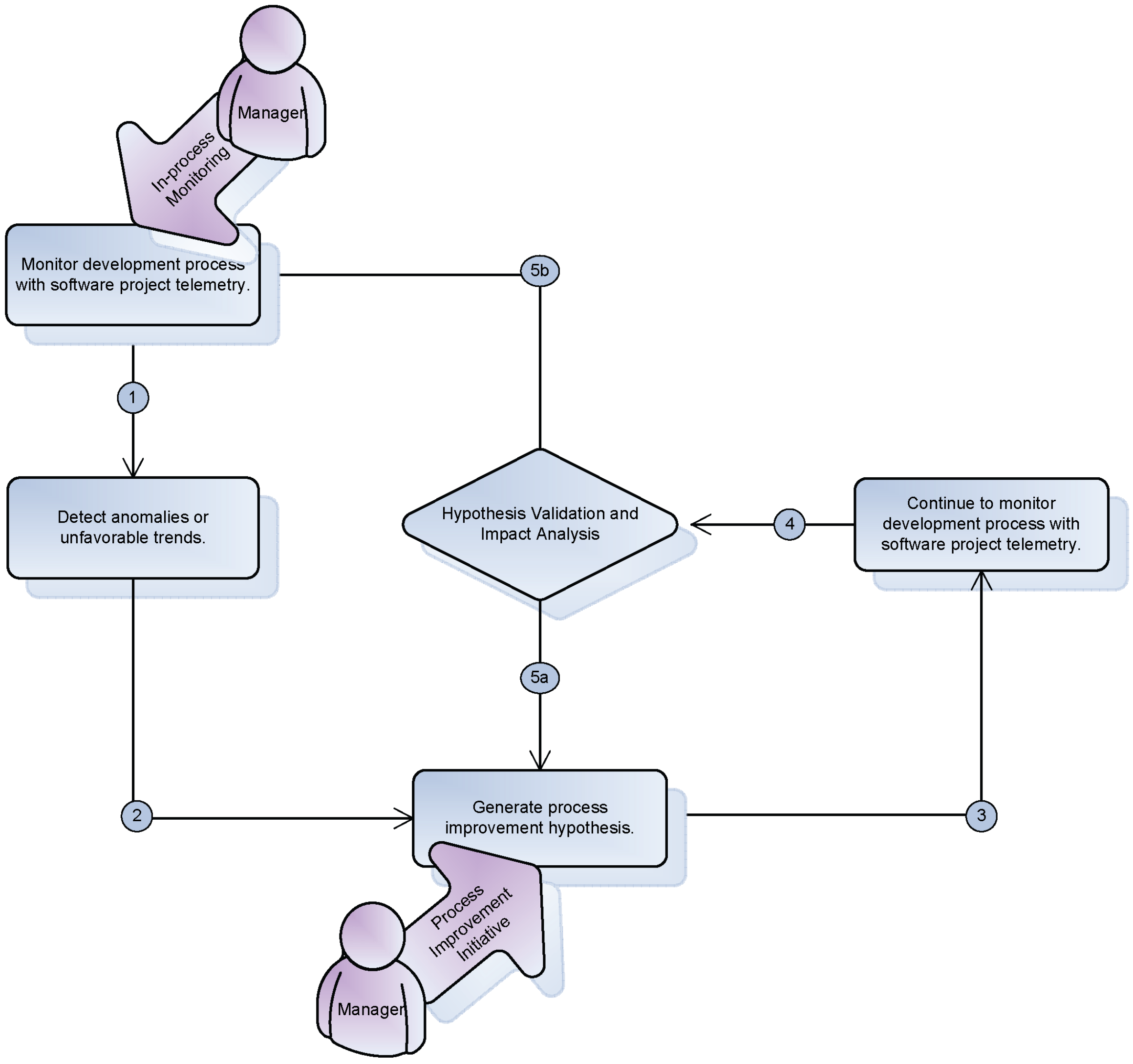

Manager
In-process Monitoring
Monitor development process with software project telemetry.
5b
1
Detect anomalies or unfavorable trends.
Hypothesis Validation and Impact Analysis
4
Continue to monitor development process with software project telemetry.
5a
2
Generate process improvement hypothesis.
3
Process Improvement Initiative
Manager

Figure 2.1. Telemetry-based Process Improvement

Software project telemetry operates primarily in two modes: (1) in-process project monitoring mode, and (2) process improvement mode. The two modes are closely related, and sometimes indistinguishable in practical project management. However, I am trying to keep them separated here in order to make the concept clear.

The steps for in-process project monitoring mode start from the upper arrow in Figure 2.1. Telemetry streams are monitored for anomalies and unfavorable trends. If anomalies or unfavorable trends are detected, then the project manager must investigate the cause. Multiple telemetry streams, representing different perspectives on development process, are used to detect correlations. For example, one might find that complexity telemetry values are increasing as well as defect density. Since telemetry streams consist of time-stamped events, the sequence of detected changes in different telemetry streams might help the project manager generate hypothesis about the causal relationship and corrective measurements for process improvement. For example, one might identify code complexity as the likely cause for high defect density. Once the process improvement hypothesis is generated, the project manager tries corrective action such as simplifying over-complex modules, and continues to monitor telemetry streams in order to check whether the action results in a decrease in defect density. It might be necessary to monitor new telemetry streams embedded with the hypothesis at this stage. One can also monitor other telemetry streams to check if such corrective action has unintended side-effects (impact analysis). If the hypothesis is correct, it will be validated by telemetry streams. If telemetry streams indicate otherwise, then there must be other reasons that cause high defect density, and the project manager must try other corrective measures. The knowledge acquired through such exercise can be generalized into rules. The rules are stored in the software organization's knowledge repository, and used in the next project.

The steps for process improvement mode follow the same loop as the steps for in-process project monitoring mode. The only difference is that they start from the lower arrow in Figure 2.1, and implementation of process improvement measures does not have to wait until anomalies or unfavorable trends are detected in telemetry streams.

2.6 Summary

Software project telemetry includes both (1) highly automated measurement machinery for metrics collection and analysis, and (2) a methodology for in-process, empirically-guided software development process problem detection and diagnosis. It overcomes many of the difficulties in existing approaches to software measurement and process improvement. The next chapter compares and contrasts software project telemetry to those approaches.

Chapter 3

Related Work

To understand how software project telemetry relates to other research, it is useful to think in terms of two concepts: *measurement machinery* and *process methodology*. Measurement machinery refers to how software metrics are collected and analyzed. In software project telemetry, sensors collect metrics automatically and unobtrusively. Metrics are abstracted into telemetry streams, charts, and reports, representing high-level perspectives on software development. Project management and process improvement decisions are based on trends detected in telemetry streams. Process methodology refers to specific techniques used to improve the quality of software development effort. Software project telemetry employs cycles of process problem detection, improvement hypothesis generation, change implementation, and hypothesis validation to empirically guide project management and process improvement decision-making.

This chapter compares and contrasts software project telemetry to other approaches to measurement machinery and process methodology. Some approaches, such as the Personal Software Process (PSP) [41, 42], can be compared to software project telemetry with respect to both measurement machinery and process methodology. Other approaches, such as the Constructive Cost Model (COCOMO) [10, 11], are only comparable with respect to measurement machinery. Still others, such as the Goal-Quality-Metric paradigm (GQM) [9, 8], and the Software Capability Maturity Model (CMM) [67, 75], are only comparable with respect to process methodology.

This chapter proceeds with an overview of software measurement theory in Section 3.1, which serve as the foundation of any software measurement programs. Section 3.2 discusses the Personal Software Process. Section 3.3 discusses process prediction models, especially the Constructive Cost

	Internal Attributes	**External Attributes**
Software Product	size, complexity, cohesion, coupling, etc.	quality, reliability, maintainability, portability, etc.
Software Process		time, effort, cost, etc.

Table 3.1. Software Measurement Classifications

Model. Section 3.4 discusses the Goal-Question-Metric paradigm. Section 3.5 discusses maturity frameworks, especially the Software Capability Maturity Model. Section 3.6 concludes the chapter with a summary.

3.1 Software Measurement Theory

As noted by DeMarco [25], *You can neither predict nor control what you cannot measure*. Measurement is the first step of transforming software engineering from an art where the success of a project depends on competence and commitment of individual developers, to a scientific discipline where project outcome is both predictable and controllable.

Measurement is "the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules" [31]. This definition depends on two related concepts: *entity* and *attribute*. An entity can be a physical object such as a program, an event such as a release milestone, and an action such as testing software. In software measurement, entities are usually divided into 2 categories: *software product* and *software process*. An attribute is a property of the entity, such as the size a program, the size of the test scripts, and the time required to finish a milestone. Attributes are generally divided into 2 categories: *internal attributes* and *external attributes*. Measures for internal attribute can be computed based on the entity itself; while measures for external attribute depends on both the entity and the environment in which the entity resides. The resulting classification scheme is depicted in Table 3.1.

The representation theory formalizes the process of mapping from an empirical relation system to a numerical relation system. Under this theory, a "measure" is nothing but the number assigned to describe some attribute of an entity by the mapping process. However, the theory does imply that not all mappings are the same. For example, the result of a sports competition, the first place, the second place, and the third place are usually mapped to real number *1*, *2*, and *3* respectively. Since sports competition result contains only ordinal information, it is equally valid to use *1*, *10*, and *100* as the mapping result. It is meaningless to add these numbers. This simple example shows that not all valid mathematical analyses in numerical relation system are valid in the original empirical relation system.

Formally, the measurement mapping together with the associated empirical and numerical relation systems is referred to as the "measurement scale". It is the measurement scale that determines valid mathematical operations that can be performed. In general, people classify measurement scale into 4 types with increasing level of restrictiveness: *nominal*, *ordinal*, *interval*, and *ratio*.

- *Nominal Scale* — The empirical relation system consists only of different classes. An example is the type of software fault, such as *specification*, *design*, and *coding*. There is no notion of ordering between different classes. As a result, any distinct numbering representation is a valid measure.
- *Ordinal Scale* — It preserves ordering. The empirical relation system consists of classes that are ordered. An example is defect severity, such as *minor*, *major*, and *critical*. Any mapping that preserves the ordering is a valid mapping, and the numbers represent ranking only. Arithmetic operations, such as *addition*, *subtraction*, *multiplication*, and *division*, have no meaning.
- *Interval Scale* — It preserves not only ordering but also differences. The difference between any two of the ordered classes in the range of the mapping is the same. Only *addition* and *subtraction* are valid. For example, when talking about time, we can say that "year 2000 is 1000 years later than year 1000", and "year 3000 is 1000 years later than year 2000", but we cannot say that "year 2000 is twice as late as year 1000".
- *Ratio Scale* — It preserves ordering, differences, and ratios. The measurement starts from a zero element representing total lack of attribute, and increases at equal intervals know as units. All arithmetic operations, *addition*, *subtraction*, *multiplication*, and *division*, can be meaningfully applied. Using the length of software code as an example, we can say that "this code contains no lines", "this code contains 20 more lines than that code", and "this code contains twice as many lines as that code".

In the current implementation of software project telemetry, mathematical manipulation of software metrics occurs at two consecutive stages: *reduction processing* and *telemetry language processing*.

Reduction processing is the process of generating basic telemetry streams by filtering, synthesizing, and aggregating raw metrics collected by sensors. Reduction functions implement different reduction behaviors. They form the lowest level, atomic “build blocks” of the software project telemetry infrastructure observable by an end user. Though data points in telemetry streams are mapped to real numbers by reduction functions, they can be of any measurement scale in theory. The reduction process itself is treated as a black box by the infrastructure, and it is the responsibility of individual reduction function implementation to ensure that sensor data are manipulated in a meaningful way.

Telemetry language processing acts on telemetry streams. The data points in telemetry streams are treated as if there were of ratio scale by the language interpreter. As a result, the language allows addition, subtraction, multiplication, and division between telemetry streams.

Theoretically, there is possibility of scale type mismatch between reduction processing and telemetry language processing. Telemetry language processing assumes that all data points in telemetry streams are of ratio scale, which cannot be guaranteed by reduction processing. This could possibly lead to problems, because telemetry language might allow meaningless mathematical operations to be applied. The problem could be solved by introducing measurement scale type system into the telemetry language, requiring all reduction functions to tag telemetry streams with scale types, and doing additional check during language interpretation. However, doing so complicates the language design and its implementation. Currently, software project telemetry takes a more pragmatic approach by relying on end users to use telemetry language in a meaningful way. In the future, the usage pattern of the language needs to be studied in order to determine the cost and benefit of introducing scale type system into the language.

3.2 The Personal Software Process

The Personal Software Process (PSP[1]) [41, 42] is a self-improvement process for software developers, and a ground-breaking approach that adapts organizational-level software measurement and analysis techniques to the individual.

The PSP provides both measurement machinery and process methodology. The primary goals of the PSP are to improve project estimation and quality assurance. The goals are pursued by

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

observation-evaluation-modification cycles. Developers observe their performance by recording how they develop software. They record the amount of time they spend, the size of the work product, and the defects they made while developing software. At the end of each project, developers evaluate how they performed by conducting standard analyses on the metrics they collected. Based on project postmortems, developers gain insight into their development process, and modify it in an attempt to improve it. A new cycle starts with the modified developement process.

The original PSP proposed by Humphrey uses a manual approach, which is very tedious. For instance, every time a compilation error occurs, the developer has to stop his/her current work, and log on paper forms the details about the error. Though several studies [32, 35, 53] have shown that the PSP appears to help improve software development, the anecdotal evidence suggests that the overhead involved in manual data collection affects its adoption. For example, a report on a workshop of PSP instructors [12] reveals that in one course of 78 students, 72 of them abandoned the PSP because they felt “it would impose an excessively strict process on them and that the extra work would not pay off”. None of the remaining 6 students reported any perceived process improvements. Moreover, manual data collection is susceptible to bias (either deliberate or unconscious), error, omission, and delay. A study [50] of the data collected in the PSP showed that there were significant issues of data quality, and the combination of data collection and analysis errors called into question the accuracy of manual PSP results. Humphrey, the author of PSP, also admits in his book *a Discipline for Software Engineering* [41] that “it would be nice to have a tool to automatically gather the PSP data”.

Tools, such as LEAP [62], PSP Studio [36] and Software Process Dashboard [82], does exist to support the original manual PSP. These tools follow the same approach to user interaction by displaying dialog boxes where the user can log effort, size, and defect information. Though tool support lowers data collection overhead considerably, it turns out that the adoption of these tools is not satisfactory because of the requirement that the user constantly switch back and forth between doing work and telling the tool what work is being done [49] [51]. This chronic context switch appears to be a problem for most developers.

Software project telemetry uses sensors to collect metrics.[2]. Sensors are attached to software development tools, which monitor some form of state change in the project development environment. Sensors intend to collect metrics automatically and unobtrusively in order to keep data collection overhead sufficiently low, so that developers are not distracted from their primary tasks – developing software products. Compared to manual and tool-based data collection, sensor-based approach not only automates metrics collection in an unobtrusive manner, but also eliminates the chronic context-switch overhead. Details about sensor data collection, as well as its restriction, are discussed in Section 2.2 of Chapter 2.

As far as process methodology is concerned, the PSP uses observation-evaluation-modification cycles to improve software development process. One cycle corresponds to the life time of one project, and process improvement is based on comparison of different projects. This is essentially model-based cross-project comparison. The limitation is that it requires a historical database of finished projects. The PSP does not yield benefit unless such a database is accumulated first. For example, one of the practices of the PSP is to use statistical regression to predict project time based on planned project size. This requires sufficient number of data points with respect to time and size information of past projects. Even if the accumulation of a historical project database is not a problem, the PSP user still must make sure that the context of the current project is consistent with the contexts of the finished projects in the project database. Otherwise, the prediction process is like comparing apples to oranges. The context consistency problem will be discussed in detail Section 3.3 of this chapter, since all model-based approaches face the same limitation.

Similar to the PSP, software project telemetry uses cycles to improve software development process. The cycle includes process problem detection, hypothesis generation, change implementation, and hypothesis validation. The difference is that software project telemetry cycle does not correspond to the life time of a project. It involves much smaller time scale, and a single project can have multiple cycles. The idea behind software project telemetry is that comparison can be made between two different periods of the same project, instead of between two different projects, and the changes in the development process and their trends can be used as the basis for process improvement decision making. Since software project telemetry does not make model-based cross-project

[2]Sensor-based approach to metrics collection is pioneered in the Hackystat project [51], developed in Collaborative Software Development Lab (CSDL) of University of Hawaii. Hackystat is a framework that provides novel support for software measurement definition, validation, and software engineering experimentation. I have been on the core Hackystat development team since 2002, while doing software project telemetry related research. The reference implementation of software project telemetry mentioned in this paper is based on the Hackystat framework.

comparison, there is neither need to accumulate historical project database, nor necessity to ensure context consistency between different projects.

3.3 The Constructive Cost Model

The Constructive Cost Model (COCOMO) [10, 11] is a model for software project cost / effort estimation. It belongs to the branch of software engineering research called model-based process prediction. This section begins with the more general topic of model-based process prediction before going to the details of COCOMO.

The research in the area of model-based process prediction typically involves the following basic procedure: (1) collect a set of process and product metrics, such as size, effort, complexity, and defects, for a set of completed software projects, (2) generate a model to fit the observed data, (3) and claim that the model can 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 model might predict that the future implementation of a module with complexity *C* will be prone to defects with density *D*.

Model-based process prediction can be compared to software project telemetry with respect to measurement machinery. The difference is that prediction in software project telemetry does not require the building of a model. Instead, it relies on changes in the development process and their trends to make short-term in-process predictions. The predictions made in software project telemetry and those made in model-based approaches tend to be of a different nature: model-based approaches tend to make end-point estimations (predictions for all phases of a software project as a whole), such as X man-hours are needed to finish project A; while software project telemetry tend to make in-process predictions, such as the number of open bugs in system B will continue to increase if system test coverage does not stop dropping.

Since model-based process prediction researches follow similar approaches in model building and process prediction, COCOMO is used to illustrate how they work. COCOMO is chosen because it is one of the widely available and accepted models in the public domain. COCOMO is developed by Barry Boehm and his associates at University of Southern California. The model estimates effort and schedule required to complete a software project. The COCOMO 81 was the original model published in the book *Software Engineering Economics* [10]. It offers 3 levels of model with increasing detail and accuracy: *basic*, *intermediate*, and *detailed*. COCOMO II [11] is an updated version of the original model to reflect the changes in software development practice. Like the first version, COCOMO II offers 3 levels, *application composition*, *early design*, and *post-architecture*, to explicitly model the fact that uncertainty of effort and schedule estimates decrease through software project life cycle.

The post-architecture model is used below to illustrate how COCOMO works. The estimation equations in the post-architectural model are:

$$PM = A * \prod_{i=1}^{17} EM_i * Size^{(B+0.01*\sum_{i=1}^{5} SF_i)} \tag{3.3.1}$$

$$TDEV = C * PM^{(D+0.002*\sum_{i=1}^{5} SF_i)} \tag{3.3.2}$$

where PM is estimated effort in person-months. A person-month is the amount of time one person spends working on a software development project for one month. Note that this is in nominal terms, which does not take schedule compression or expansion into account[3]. $Size$ is the primary input to the model. It is expressed in thousands of source lines of code (KSLOC). COCOMO II not only offers detailed rules on how to count lines of code, but also provides methods to covert other counting results, such as function points[4] [2] and object points [5, 6], to lines of code. $TDEV$ is the amount of calendar time it will take to develop the software product. The average number of staff can be derived by dividing $TDEV$ from PM.

A, B, C, D, SF_i and EM_i are all constants in the model. SF_i is called scale factor which influences effort exponentially. Scale factors are used to account for the relative economy or dis-economy of scale encountered for software projects of different sizes. EM_i is called effort multiplier which influences effort multiplicatively. Effort multipliers are used to adjust for different product, project, platform and personnel factors in different software product development. Both effort multipliers and scale factors are defined by a set of rating levels (e.g. Very Low, Low, Nominal, High, etc.).

[3]COCOMO II offers an effort multiplier $SCED$, which can be used to adjust for the effect of schedule compression or expansion.

[4]Function point is a measure of program size independent of technology and programming language. The value of function point FP is the product of unadjusted function points UFP and technical correction factor TCF. Support for setting up function point analysis program is available from International Function Point User Group (http://www.ifpug.org).

Every few years, COCOMO team updates the model by supplying new calibration values for the constants to reflect latest change in software production practice in industry. For example COCOMO II 2000 post-architecture model calibration values were obtained by calibrating the model to the actual parameters and effort values for the 161 projects in the COCOMO II database at that time. These values represent the software industry average. COCOMO recommends its users to calibrate A, B, C and D to their local development environment in order to increase prediction accuracy of the model.

COCOMO enjoys wide acceptance in both academia and industry. Various extensions have been developed since the publication of the original model. These extensions include COQUALMO (Constructive Quality Model) [15], COCOTS (Constructive COTS Model) [1], and CORADMO (Constructive Rapid Application Development Model) [30]. Commercial implementations include Costar from Softstart Systems [20], Cost Xpert from Cost Xpert Group Incorporated [83], and Estimate Professional from Software Productivity Center Incorporated [70].

The basic idea behind COCOMO and other process prediction models is cross-project comparison. Unfortunately, there are a number of difficulties in adopting this method.

First, model-based process prediction assumes that the software organization has a relatively stable and repeatable development process. However, according to a Software Engineering Institute (SEI) survey of 542 software development organizations [68], 67% of them are at CMM maturity level 1 - the lowest maturity level. By definition, the software processes at level 1 are *ad hoc* and sometimes *chaotic*, and they change as work changes. As a result, it is generally impossible to make predictions for organizations at this level. In other words, two-thirds of software organizations cannot benefit from model-based process predition, such as COCOMO.

Second, the prediction power of the models highly depends on how well model calibration is performed. This can be thought of as a context consistency problem. In order to use the model, practitioners must confirm that the set of projects used to calibrate the model are similar to the project they wish to predict. Otherwise, they must recalibrate the model using the data in the organization's historical project database. This involves replicating the model-building method within the practitioner's organization, with the risk that the organization may have already changed and the context of future projects may differ from those in the historical project database.

Lastly, model-based process predictions are primarily designed to be used at very early stage of a software project, or even before a project actually starts. Therefore, they tend to make end-point estimations (i.e. the prediction is made for all phases of the project as a whole). For example, COCOMO estimates that 586 person-months are required to develop a software with estimated size of 100 KSLOC[5]. But when 300 person-months have been spent writing 60 KSLOC, the model does not give any indication whether the project will still be on-target or not. One will know the answer only after the entire project is finished, but everything will be too late at that time. In other words, end-point estimations have no in-process control capabilities.

Software project telemetry avoids the above-mentioned difficulties by shifting the focus of process prediction. It makes no attempt to build a rigorous cross-project comparison model in order to make a prediction before the project starts. Instead, it employs a more agile approach to compare software processes in different periods within the same project. It relies on the changes in the development process, as well as the trends of the changes, to make short-term predictions for the purpose of in-process project management.

3.4 The Goal-Question-Metric Paradigm

While software project telemetry requires telemetry streams to be used to monitor the development process of a software project, it never specifies what telemetry streams should be used. This is actually not a deficiency of software project telemetry. Because different projects and different organizations have different development environment and different project goals, it is impossible to prescribe the required telemetry streams.

Many organizations opportunistically collect metrics based on what is easy to collect. If software project telemetry is introduced to these organization, they would naturally extend the bottom-up approach by generating telemetry steams based on available metrics. While bottom-up approach sometimes works, it is best to collect data and use software project telemetry in a top-down fashion with a clear purpose in mind.

The Goal-Question-Metric paradigm (GQM) [9, 8] has no measurement machinery, but it provides a top-down, goal-oriented process methodology that can be used in conjunction with software project telemetry. In GQM, software measurement is driven by high level goals. Usually business goals of an organization are formed first and then translated into improvement goals of software

[5]Suppose all scale factors and effort multipliers take the rating of *nominal*. Using COCOMO II 1997 calibration data, the estimation equation is $PM = 2.94 * Size^{1.15}$.

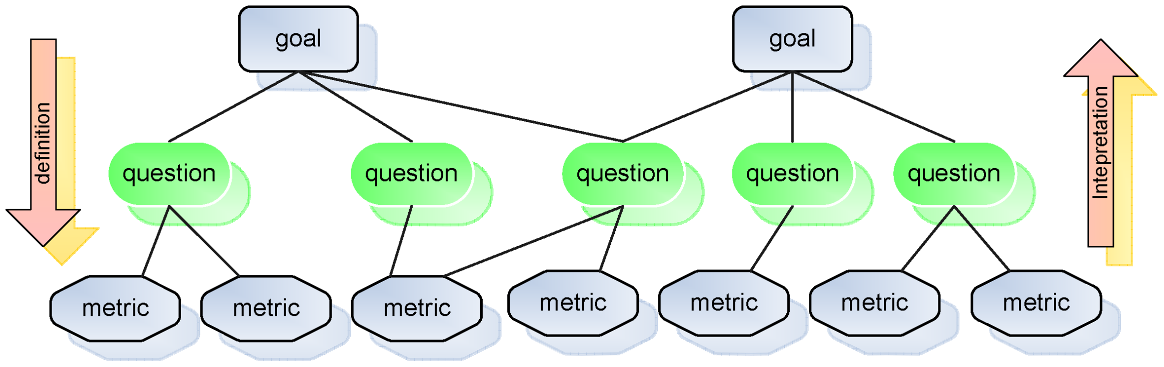


Figure 3.1. Goal-Question-Metric Paradigm

development, which, in turn, are translated into measurement goals. A metrics program is used to fulfill the measurement goals. Based on the measurement outcome, the organization can generate hypotheses and make proper decisions in order to reach the improvement goals and finally the business goals.

GQM offers a top-down method to translate high level measurement goals into questions that need to be answered in order to reach the goals. Based on the questions a set of relevant metrics can be identified and collected, which provide answers to those questions. This leads to a tree-structured graph depicted in Figure 3.1.

GQM measurement goal is stated in 5 dimensions: *study object*, *purpose*, *quality focus*, *view point*, and *environment*. A concrete example can be found in [77, 76]. The authors studied causes and effects of interrupts on software development work, and their measurement goal was "*analyze* interrupts and their effects *for the purpose of* understanding *with respect to* impact on schedule and the cost-benefit of interrupts *from the viewpoint of* project team *in the context of* project X".

GQM is well-known. Case studies of successful experience abound [7, 56, 33]. The key to the success of GQM implementation appears to be the establishment of well-defined measurement goals and the derivation of software metrics that can be used to provide useful information to meet the goals.

The main drawback of GQM paradigm is the lack of attention to the actual measurement process. Metrics collection, interpretation and analysis are not a part of the paradigm. Implicitly, GQM assumes that once all required metrics are identified, the rest steps (metrics collection and analysis) would be easy.

However, the drawback can be overcome by using GQM in conjunction with software project telemetry. When used together, they reinforce each other: GQM defines useful metrics and telemetry streams, and relates them to software organization's business, project, product, and process goals; while software project telemetry provides an automated infrastructure for metrics collection and analysis, as well as an in-process, empirically-guided methodology for process improvement. For example, "continuous GQM" tries to implement GQM in an automated way in which data is collected, analyzed, and presented automatically with minimal human effort [59].

3.5 The Software Capability Maturity Model

The Software Capability Maturity Model (CMM) [67, 75] is a process maturity framework developed by Software Engineering Institute (SEI). Other similar frameworks include ISO 9001 [44] and SPICE and ISO/IEC 15504 [45, 29]. They share common properties, such as using metrics as means to help software organizations improve their development process, and to assess their capabilities. In these approaches, an externally defined reference framework is used to prescribe the activities, methodologies and practices a software organization should implement. The implicit assumption is that the prescribed processes are needed by most organizations in order to deliver high quality software in a repeatable and controllable manner, and a mature software development process will deliver high quality software products on time and within budget. Process assessment is used to compare organizational processes with the reference framework, which serves as an effective driver for process improvement. The assessment can be done by the software organization itself, by the second party, or by an independent third party. Based on the assessment results, organizations can find directions for process improvement.

Software project telemetry is closely related to process maturity. On one hand, higher process maturity offers greater visibility into development activities. Since we can only measure what is visible in a process, higher process maturity means more telemetry streams can be generated and monitored. The process maturity scales offer a convenient context for determining what telemetry streams to use first, and how to plan software project telemetry program so that it grows to embrace additional aspects of development and management. On the other hand, software project telemetry offers a methodology for process improvement based on quantitative feedback from existing development processes. It is especially helpful for software organizations to achieve high process maturity levels where quantitative measurement is mandated, such as CMM level 4 or 5.

The following discussion will focus on CMM to illustrate how a maturity framework works, since most maturity frameworks work in similar ways.

The goal of CMM is to determine whether a software development organization has a sound management infrastructure, and to assess its level of competence in building high quality software products. CMM is a staged model, which provides a set of requirements that software development organizations can use to set up software processes to control software product development. It ranks a software organization's process capability on a maturity level from 1 (lowest) to 5 (highest):

1. *Initial Stage:* The software process at this level is characterized as *ad hoc* and sometimes *chaotic*. Success of software projects depends on competence and commitment of individual developers. Few software processes are defined, and they change as work progresses. As a result, schedules, budgets and quality are generally unpredictable.

2. *Repeatable Stage:* Basic project management processes are in place. Software organizations at this level have controls over software requirements, project planning and tracking, configuration management, quality assurance and subcontractor management. They are able to track project cost and schedule, and they can repeat earlier successes on projects with similar applications.

3. *Defined Stage:* The software processes for both management and engineering activities are documented, standardized and integrated into a standard software process for the whole organization. All projects use an approved, tailored version of the organization's standard software process to develop and maintain software.

4. *Managed Stage:* Detailed measurement programs are in place to assess software development processes and product quality. Both software process and products are quantitatively understood and controlled. Software organizations at this level are able to tailor development processes to specific needs with predictable outcomes.

5. *Optimizing Stage:* Continuous process improvement is enabled by quantitative feedback from software development processes and from piloting innovative ideas and technologies.

Each maturity level is associated with a number of processes that an organization must implement. These processes are grouped into key process areas (KPA). Each KPA has a set of goals, capabilities, key practices, as well as measurements and verification practices. There are a total of

CMM Levels	**Key Process Areas**
Level 1 – Initial	None.
Level 2 – Repeatable	Requirements Management, Software Project Planning, Software Project Tracking and Oversight, Software Subcontract Management, Software Quality Assurance, Software Configuration Management.
Level 3 – Defined	Organization Process Focus, Organization Process Definition, Training Program, Integrated Software Management, Software Product Engineering, Intergroup Coordination, Peer Reviews.
Level 4 – Managed	Quantitative Process Management, Software Quality Management.
Level 5 – Optimizing	Defect Prevention, Technology Change Management, Process Change Management.

Table 3.2. CMM Levels and Key Process Areas

52 goals and 150 key practices. Some are related to setting up basic project management controls; some are aimed at establishing an infrastructure that institutionalizes effective software engineering and management processes across projects; while the rest are focused on performing a well-defined engineering process that integrates all software engineering activities to produce correct, consistent software products effectively and efficiently. The maturity level of a software organization is determined by its demonstrated capability in the key process areas associated with that level. Table 3.2 lists CMM levels and their associated key process areas.

CMM has received much attention in both academia and industry. Quite a few positive experiences have been reported in the literature [43, 27, 24, 26]. For example, Humphrey *et. al.* [43] reported a software process improvement program at Hughes Aircraft with estimated annual saving at about $2 million.

CMM makes use of software metrics to help software organizations improve their development process. It prescribes that in all key process areas measurement should be taken to determine the status of development activities. However, it does not prescribe explicitly how the measurement processes themselves should be implemented. In fact, Humphrey is aware of the difficulty of implementing a measurement program in a software organization. He mentioned in [75] that “the greatest potential problem with the managed process (CMM level 4) is the cost of gathering data. There are an enormous number of potentially valuable measures of the software process, but such data is expensive to gather and to maintain.”.

CMM Maturity Level	**What to Measure**
Level 1 – Initial: ad hoc	baseline
Level 2 – Repeatable: process dependent on individual	project
Level 3 – Defined: process defined and institutionalized	product
Level 4 – Managed: process measured quantitatively	process and feedback for control
Level 5 – Optimizing: continuing improvement based on measurement	process and feedback for changing process

Table 3.3. Process Maturity and Measurement

Software project telemetry is a nice complement to the void left by CMM. It provides not only an automatic and unobtrusive way of gathering process data, but also a methodology of using quantitative data to analyze and modify software development process in order to prevent problems and improve efficiency. It is especially helpful to software organizations at high process maturity levels, such as CMM level 4 or 5.

CMM is useful to software project telemetry too. CMM maturity levels provide a convenient context for determining what telemetry streams to use first, and how to plan software project telemetry program so that it grows to embrace additional aspects of development and management. Indeed, several authors have studied the relationship between software measurement and process maturity. For example, Table 3.3 lists Pfleeger and McGowan's recommendation of collecting different measures depending on a software organization's maturity level [69].

3.6 Summary

This chapter compares and contrasts software project telemetry to other metrics-based approaches. The Personal Software Process suffers from chronic metrics collection and analysis overhead. The Constructive cost model not only assumes that software development process is stable and thus predictable but also requires calibration. The Goal-Question-Metric Paradigm offers a high-level abstract process methodology but lacks attention to the actual measurement process. The Software Capability Maturity Model prescribes that measurement be taken to assess the status of development activities but does not specify how the measurement process should be implemented. Software project telemetry addresses these limitations through automated metrics collection and intuitive visual metrics analysis for in-process empirically-guided project management and process improvement. The next chapter details an implementation of software project telemetry.

Chapter 4

System Implementation

This chapter describes the internal details of my implementation of software project telemetry. The target audience is developers who wish to maintain and improve the telemetry core components. For developer who wants to implement additional telemetry reducers or end users, resources listed in Table 4.1 might be more useful.

The software project telemetry system is designed and implemented by me as part of my dissertation research. It is built on top of the Hackystat foundation leveraging its generic framework of metrics collection and project management. Hackystat is a framework that provides novel support for software measurement definition, validation, and software engineering experimentation. It is developed in Collaborative Software Development Lab (CSDL) at University of Hawaii.

Role	**Resource**	**Link**
End User	Software Project Telemetry End User Guide; Software Project Telemetry Language Specification.	Appendix B Appendix A
Telemetry Reducer Developer	Software Project Telemetry Reducer Developer Guide.	Appendix C
Telemetry System Maintainer	System Implementation; Software Project Telemetry Language Specification.	This Chapter Appendix A

Table 4.1. Software Project Telemetry Resources

In a sense, the software project telemetry system can be viewed as an extension to the functionalities of the Hackystat framework. CSDL members sometimes use “Hackystat” as an umbrella term to refer to Hackystat infrastructure plus all extensions built on top of it. However, for the sake of clarity, this dissertation makes the distinction between Hackystat framework and the software project telemetry system. Throughout this dissertation:

- *Hackystat framework* — refers to the infrastructure,
- *Software project telemetry system* — refers to my implementation of the telemetry concept utilizing that infrastructure.

The reason that I chose to base my implementation of software project telemetry on the Hackystat framework instead of building everything from scratch is because:

1. Hackystat uses sensors to collect software metrics, which is an essential prerequisite for *software project telemetry*. Its metrics collection subsystem is flexible, and new sensors can be easily plugged in.
2. Hackystat offers sophisticated support for metrics data storage, processing, and presentation.
3. Hackystat has flexible extension mechanism. New analyses and alerts can be easily plugged in.
4. I have been on the development team of Hackystat since 2002 and have extensive knowledge of the framework.

4.1 Implementation Details

The actual implementation details of software project telemetry are omitted in this proposal. Interested readers are encouraged to examine the following documents:

- *Telemetry Language Specification*:
  Appendix A
- *Telemetry End User Documentation*:
  http://hackystat.ics.hawaii.edu/hackystat/docbook/ch05.html

- *Telemetry Developer Guide*:
  Appendix C

- *Complete Source Code*:
  http://hackydev.ics.hawaii.edu/hackyDevSite/lastBuild.do
  Please note that the source code is updated daily. Please follow the link "Build Results" – "java2html", and the major portion of telemetry related implementation code is in the module "hackyTelemetry".

When the dissertation is complete, this chapter will be organized into the following sections. Section 4.1 will give a brief introduction of Hackystat architecture. Section 4.2 will detail *software project telemetry* related implementation.

4.2 Time Frame

The implementation is complete. Two instances of the software project telemetry system are deployed. One is on the public Hackystat server which is used in the case studies in Chapter 6 and 7. The other instance is deployed on Ikayzo server which is used in the case study in Chapter 8.

Chapter 5

Research Designs and Evaluation Strategies

There are a variety of possible approaches to empirical evaluation of the claims made in this dissertation. This chapter provides an overview based on the book *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* [21] by J. W. Creswell. My approach to the evaluation of software project telemetry is based on concepts and techniques presented in that work.

This chapter starts with a brief review of approaches to evalution in Section 5.1 followed by a discussion of software project telemetry with respect to its evaluation strategies in Section 5.2. The evaluation of software project telemetry is carried out in 3 case studies. Section 5.3 gives a brief overview of each case study.

5.1 Review of Research Designs

Creswell [21] categorizes research methods into 3 paradigms: quantitative, qualitative, and mixed methods. Each paradigm has different philosophical assumptions about what constitutes knowledge.

The quantitative paradigm is related to *postpositivism*[1], which is often labeled as "the traditional way of doing scientific research". Postpositivism seeks to develop relevant true statements that can

[1] *Postpositivism* differs from *positivism* by recognizing there is no absolute truth. A hypothesis can never be proved. It is accepted by failing to reject it.

explain a situation or describe a causal relationship. The research is the process of making claims and using data and evidence to test hypotheses.

The qualitative paradigm is related to *constructivism*. It assumes that individuals seek understanding of the world in which they live to develop subjective meanings of their experiences. Qualitative research relies much on the participants' views of the situation being studied. It tends to use open-ended questions to allow the participants to express their views.

The mixed methods paradigm is related to *pragmatism*, in which knowledge claims arise out of actions, situations, and consequences rather than antecedent conditions (such as in postpositivism). Instead of methods being important, the problem itself is the most important. As a result, mixed methods researchers do not subscribe to any single system of philosophy. They use both quantitative and qualitative data to best understand the problem and find solutions.

5.1.1 Quantitative Research Paradigm

A quantitative paradigm is one in which a researcher uses postpositivism approaches to acquire knowledge. The knowledge is conjectural in nature. The researcher tests a theory by specifying narrow hypotheses, collecting closed-ended data on predetermined instruments, and using statistical procedures to analyze the data to either support or refute the hypotheses. Typical inquiry strategies are surveys and experiments.

A survey can provide quantitative description[2] of trends, attitudes, or opinions of a population by studying a sample of that population. The purpose is to generalize the results, so that inferences can be made about some characteristic, attitude, or behavior of the population [4]. The main advantage of a survey is the economy of design and the possibility of rapid turnaround in data collection.

In an experiment, a researcher may also select a sample and generalize the results to a population. However, the basic intent of an experiment is to test the impact of a treatment on an outcome, controlling for all other factors that might influence that outcome. There are many different experimental designs, ranging from true experiments (with random assignment of subjects), quasi-experiments (non-randomized designs), to correlation studies.

[2]A survey can also gather qualitative information depending on the questions it asks.

5.1.2 Qualitative Research Paradigm

A qualitative paradigm is one in which a researcher makes knowledge claims based primarily on constructivism perspectives, such as multiple meanings of individual experiences, and socially or historically constructed meanings. The paradigm has its roots in cultural anthropology and American sociology.

The intent of qualitative research is to understand a particular situation, event, role, group, or interaction from the views of the participants. As a result, a qualitative researcher collects data through open-ended questions or by observing the participants' behaviors. The research is largely an investigative process where the researcher gradually makes sense of a phenomenon by contrasting, comparing, replicating, cataloging and classifying the objects of study [61].

A major factor that distinguishes qualitative methods from quantitative methods is that a researcher is not prescribing the questions that need to be answered from his/her standpoint. Instead, the researcher tries to learn from the participants. Other unique characteristics include:

- Qualitative research takes place in natural settings where human behavior and events occur.
- Qualitative research is based on assumptions very different from quantitative research. Theories or hypotheses are established *a posteriori* instead of *a priori*.
- Qualitative research uses multiple interactive methods which includes open-ended questions, observations, and interviews.
- Qualitative research focuses on the participants' perceptions and experiences, as well as their understanding of the world in which they live and work.
- Qualitative research is emergent rather than tightly prefigured.
- Qualitative data are usually descriptive, in the form of text rather than numbers. They are fundamentally interpretive.

A primary reason for conducting a qualitative study is that the research is exploratory in nature. The researcher seeks to listen to the participants in order to build an understanding based on their ideas and views. Ethnography and case study are the two commonly used methods.

In ethnography, the researcher studies an intact group of people in a natural setting over a prolonged period of time. The intent is to obtain a holistic picture of how people describe and structure their world by observing and interviewing them. The research process is flexible and typically evolves contextually in response to the lived realities encountered in the field setting [57].

In a case study, the researcher explores in depth a program, an event, an activity, a process, or one or more individuals, by collecting detailed information using a variety of procedures over a sustained period of time. The case is always bounded by time and activity.

Other methods include grounded theory [79], phenomenological research [63], and narrative research [16].

5.1.3 Mixed Methods Research Paradigm

A mixed methods paradigm is one in which a researcher bases knowledge claims on pragmatic grounds. Knowledge claims arise out of actions, situations, and consequences rather than antecedent conditions. To mixed methods researchers, understanding the problem and finding the solutions are more important than methodology. As a result, they are not committed to any single methodology. They use both quantitative and qualitative data to meet their needs and purposes.

According to Creswell [21], the idea of mixing different methods probably originated in 1959 when Campbell and Fiske used multiple methods to study validity of psychological traits. They encouraged others to employ their “multimethod matrix” to examine multiple approaches to data collection in a study. This prompted others to mix methods, and soon approaches associated with field methods such as observations and interviews (qualitative data) were combined with traditional surveys (quantitative data). Important works in this field include [73, 64, 66, 14, 81].

In fact, the situation today is less quantitative versus qualitative and more how research practices lie somewhere on a continuum between the two [65]. The best that can be said is that studies tend to be more quantitative or qualitative in nature.

A major factor that distinguishes mixed methods paradigm from others is that it is problem centered and real world practice oriented. Other unique characteristics include:

- A mixed methods researcher does not mix different methods blindly. There is always a purpose for "mixing", i.e. a rationale for the reasons why quantitative and qualitative data need to be mixed in the first place.

- A mixed methods researcher is "free" to choose the methods, techniques, and procedures of research that best meet their needs and purposes, rather than subscribing to only one way (e.g. quantitative or qualitative).

- A mixed methods researcher uses both quantitative and qualitative data because they work to provide the best understanding of a research problem.

In this dissertation, I will follow the mixed methods approach to empirical evaluation of software project telemetry. The rationale is given in Section 5.2.2.

5.2 Evaluation of Software Project Telemetry

The evaluation of software project telemetry consists of 3 case studies in different settings. This section discusses general strategies and procedures. A brief overview of each case study is provided in the next section (Section 5.3). The specific details are reported in the next 3 chapters (Chapter 6, 7, and 8) respectively.

5.2.1 Bounding the Study

Software project telemetry is a novel approach to metrics-based project management and process improvement. It attempts to lower metrics collection cost by using sensors to collect metrics automatically and unobtrusively, and facilitate metrics analysis by providing high level telemetry perspectives of development process. The main thesis of this research is that software project telemetry provides an effective automated approach to in-process, empirically-guided software development process problem detection and analysis.

Software project telemetry is a young technology, and as such this evaluation is an exploratory study with the purpose of understanding how software metrics participate in project management and process improvement decision-making and how software project telemetry provides or fails to provide metrics collection and analysis support. There are many competing metrics-based approaches to project management and process improvement, such as PSP, TSP and CMM. It is not the goal of this evaluation to compare software project telemetry to other approaches. It is my belief that each approach has its own unique strengths and weaknesses, and a comparative study is best performed after a good understanding of software project telemetry has been acquired. Therefore, an important part of this evaluation is to examine the application of software project telemetry in different software development environments and find out what works, what doesn't, and how the technology can be further improved.

Software development environments are diverse, and each may have different development processes and constraints on metrics collection and analysis. For example, at one end of the spectrum, there are tightly controlled environments such as those in classroom or experimental settings where it is possible to control the development process to the finest details. At the other end of the spectrum, there are free open-source projects collaborated on by geographically dispersed volunteer developers where project leaders have little control over individual developers. In the middle are the more traditional team-oriented software development environments.

The goal of this evaluation is to study the application of software project telemetry in different development environments. I categorize different environments according to how easily the technology and practice of software project telemetry can be introduced to the development team. I have identified three different but representative locations along this dimension in which to conduct case studies:

- *Classroom* — The environment of the software engineering class taught by Dr. Philip Johnson at the University of Hawaii lies at one end of the spectrum. There is no technology adoption barrier. By curriculum design, students are required to install the sensors to collect their software product and process metrics and to use software project telemetry system to perform metrics analysis.

- *CSDL* — The environment of the Collaborative Software Development Lab at the University of Hawaii lies somewhere in the middle of the spectrum. It is typical of traditional software development environments. The project is collaborated on by a team of developers at the same location. A project manager controlls the overall direction. The use of software project telemetry as a metrics-based approach to project management and process improvement is endorsed by the manager and the developers. They are willing to try the technology but will commit to it only if it can be proved useful.

- *Ikayzo* — The Ikayzo environment lies at the opposite end of the spectrum. Ikayzo provides free hosting services to open source projects. It is similar to *sourceforge.net* except much smaller in size. Ikayzo uses software project telemetry system to provide automated metrics collection and analysis support to the hosted projects, but does not participate in decision-making of the projects (including metrics application) in any way.

The experience gained from the application of software project telemetry in these three different software development environments will paint a holistic picture which helps me better understand the technology.

5.2.2 Evaluation Methodology

I will adopt the mixed methods research paradigm, collecting and analyzing both qualitative and quantitative data, while evaluating software project telemetry. A high priority will be given to qualitative information. The justifications are provided below:

To a large extent software engineering is about how people interact with each other and interact with tools to product software products. Software project telemetry is no exception. It is a metrics-based approach to project management and process improvement. It is designed to automate tedious metrics collection tasks and provide metrics analysis support through high level telemetry perspectives on the development process. It is not designed to replace human judgment in project decision-making. An important goal of the evaluation is to understand how software project telemetry provides metrics collection and analysis support and how project managers and developers interact with the technology to make decisions. As a result, a naturalistic approach and qualitative data analysis are consistent with the overall goal of this evaluation (i.e. exploration of a phenomenon).

Quantitative information such as the actual invocation of software project telemetry analysis will be used to assist or cross-validate the interpretation of qualitative findings when appropriate. The use of multiple forms of data collection and analysis have the potential to offset the weaknesses inherent within one method with the strengths of the other method.

Mixed Methods Strategies

Creswell et al. [22] identifies 4 criteria for selecting an appropriate mixed methods inquiry approach: implementation, priority, integration, and theoretical perspective. I will structure the data collection, analysis, and validation procedures in my evaluation according to these criteria. The following discussion outlines the high-level strategies. They apply in full to the case studies in CSDL and at Ikayzo, but only partially to the case study in classroom which is designed to be a pilot study to gather students' opinions in a short period of time. Details with respect to each case study is presented in Chapter Chapter 6, 7, and 8.

- *Implementation — What is the implementation sequence of the quantitative and qualitative data collection?*

  Qualitative data will mostly cover how software project telemetry is used in project management and process improvement decision-making: what works, what does not work, and what can be improved. They will be collected through ethnographic observations, questionnaires, and/or interviews. Quantitative data will mainly cover the software project telemetry system setup and administration cost and the actual invocation of telemetry metrics analysis. Both qualitative and quantitative data will be gathered concurrently, with the exception of the case study in the classroom where a questionnaire was distributed at the end of the semester.

- *Priority — What priority will be given to the quantitative and qualitative data collection and analysis?*

  This evaluation is primarily designed as a naturalistic study since its purpose is to understand a phenomenon. Priority will be skewed toward qualitative information. Quantitative information will only be used to assist or cross-validate the interpretation of qualitative findings.

- *Integration — At what stage in the research project will the quantitative and qualitative data and findings be integrated?*

  The results of the two methods will be integrated during the interpretation phase. I will either note the convergence of the findings as a way to strengthen the knowledge claims of the study or explain any lack of convergence that may result.

- *Theoretical Perspective — Will a theoretical lens or framework guide the study such as a theory from the social science or a lens from an advocacy perspective (e.g., feminism, racial perspective)?*

I choose not to use a theoretical perspective as a lens in this study.

Data Analysis Procedures

This evaluation adopts mixed methods research paradigm with a higher priority on qualitative information. According to Schatzman and Strauss [74], qualitative data analysis primarily entails classifying things, persons, events and the properties which characterize them. As a general procedure, I will organize the qualitative data from ethnographic observations, questionnaires, and interviews into categories, identify and describe patterns and themes from the perspective of the case study participants, and then attempt to understand and explain these patterns and themes. At the end of qualitative analysis, quantitative data such as software project telemetry analysis usage information will be integrated to assist or cross-validate the qualitative findings.

Verification and Validation

In order to ensure measurement validity and internal validity, the following steps will be employed:

- *Triangulation of data* — Qualitative data will be collected through multiple sources such as my personal diary, questionnaires, interviews, and observations, and reconciled with each other. Quantitative data will be used to cross-validate the qualitative findings.
- *Member checking* — The case study participants will serve as a check throughout the analysis process. An ongoing dialog will be maintained with participants with respect to my interpretations of the data.
- *Clarification of bias* — My role in each case study and possible bias will be articulated in the report of each case study.

The primary strategy I will use to ensure external validity will be the provision of detailed descriptions of each case study so that anyone interested in transferability will have a solid framework for comparison [60]. The following steps will be employed:

- I will provide a detailed account of the focus of each case study, my role, the participant's position, and the context from which data will be gathered.

- Data collection and analysis strategies will be reported in detail in order to provide a clear and accurate picture of the methods used in this study.

5.3 Overview of the Case Studies

This section provides a brief overview of each case study. The specific details are reported in Chapter 6, 7, and 8 respectively.

As a side note, I have completed the case study in classroom, but the case studies in CSDL and at Ikayzo are in preparation stage. The detailed schedule can be found in Section 5.4.

- **A pilot case study in a classroom**

  I have completed the first case study. It was conducted with senior undergraduate level and introductory graduate level software engineering students in 2005 Spring semester at the University of Hawaii. The study was designed to collect student feedback about software project telemetry in a short period of time. The design was the simple one shot case study. The students used software project telemetry to collect and analyze their metrics while performing software development tasks during the semester. Their invocation of telemetry analysis were recorded automatically by instrumenting the telemetry system. A survey was distributed at the end of the semester to collect their opinions with respect to how software project telemetry provided metrics collection and analysis support. The survey method was chosen because of its ease of administration, rapid turn around in data collection, and guarantee of 100% response rate in the classroom setting.

  The limitation of this case study is mainly related to the classroom setting. The background of the students may not be representative of those of real world programmers, and class projects tend to be smaller in scope than real world projects. Despite the limitation, I was able to obtain significant insights from the student users. Details about the case study design and the results are reported in Chapter 6.

- **A case study in CSDL**

  I am starting the second case study. It will be conducted in Collaborative Software Development Lab at University of Hawaii, where a large scale software project (100 KSLOC) is being developed and maintained by a group of 5 to 10 active developers. The project is experiencing a significant rate of integration build failure (88 failures out of 300 builds in 2004). The goal of this case study is to apply software project telemetry process methodology to investigate and improve CSDL build process.

I will customize the software project telemetry system to analyze the metrics related to quality assurance process of the developers and provide feedback. I will employ ethnography to collect detailed daily observation data, and I will interview developers on a weekly basis to detect and understand their response to telemetry feedback information. I will also collect and compare each developer's process metrics before and after the introduction of the software project telemetry feedback mechanism. Details about this case study are discussed in Chapter 7.

- **A case study at Ikayzo**

  I am starting the last case study. It will be conducted in Ikayzo[3]. Ikayzo provides free open source project hosting services, which include source code version control, issue tracking and management, automated metrics collection and analysis, and wiki discussion. The metrics collection and analysis support is provided through software project telemetry system. Ikayzo does not participate in decision-making of the hosted projects. It cannot force the projects to adopt any metrics related technology. What it does is provide automated metrics collection and analysis support at zero overhead to the projects, and let them decide the value of the metrics. The primary focuses of this case study is to understand the adoption barrier of software project telemetry technology in an open source development environment where it is impossible to force people to use the technology. Details are discussed in Chapter 8.

5.4 Case Study Schedule

The case study in the classroom was conducted in Spring 2005.

The case study in CSDL is starting. It will last 6 months. I am customizing the telemetry analysis system to provide developer project quality assurance process feedback. The initial version was deployed on the week of Oct 9. The case study can start in a week or two once the initial version is field tested.

[3]http://www.ikayzo.org

The case study in Ikayzo is starting. It will last 6 months.

Chapter 6

Case Study in a Classroom

This chapter reports on a pilot case study with senior undergraduate and introductory graduate software engineering students in 2005 Spring semester at University of Hawaii. The study had dual purposes:

- To test drive the software project telemetry system in preparation for the next two full-scale case studies as described in Chapter 7 and 8.
- To collect the students' opinion about software project telemetry when the adoption of the technology is mandated by their instructor.

The design was the simple one shot case study. The students used software project telemetry to collect and analyze their metrics while performing software development tasks during the semester. At the end of the semester, a survey questionnaire was distributed to collect their opinions with respect to software project telemetry. The actual telemetry analysis invocation data will be used to corroborate the qualitative findings.

This chapter begins with a description of the classroom setting in Section 6.1. Section 6.2 outlines the design and strategies of this case study and offers rationale for the decisions. Section 6.3 describes the researcher's role. Section 6.4 introduces data collection and analysis procedures. Section 6.5 discuses the limitations and threats. Section 6.6 reports the results. Section 6.7 concludes the chapter with a summary of the insights learned from this case study.

6.1 Classroom Setting

The case study was conducted in Dr. Philip Johnson's senior undergraduate (ICS 414) and introductory graduate (ICS 613) software engineering classes in Spring 2005 at University of Hawaii. The two classes followed the same basic curriculum except that the graduate level one had more supplementary readings. The curriculum had two equally important components:

1. **Techniques and tools for Java-based software development** — The students in the classes were divided into teams of 2 to 4 members and worked on different projects. The techniques taught in class covered code formatting and documentation best practices, design patterns, configuration management, code review, and agile development practice; while the tools used included Eclipse (an IDE primarily for Java development), CVS (a configuration management system), Ant (a Java-based build tool), and JUnit (a Java unit test framework).

2. **Software product and process metrics collection and analysis** — The two classes required the students to collect and analyze their software process and product metrics while performing software development tasks. The purpose was to help them acquire hands-on experience in collecting, analyzing, and interpreting software metrics.

The students' product and process metrics were collected and analyzed using the software project telemetry system. The implementation of the system was based on the Hackystat framework. It was first incorporated in Dr. Philip Johnson's software engineering class in 2004 Fall semester. This was the second semester that it was used. The system was deployed on a university public server[1]. It had a web interface to allow the students to manage their projects and invoke telemetry analysis over their data to gain insights into their software development processes. In order to gather product and process metrics, the students were required to instrument their development environment and build tools with sensors. These sensors collected a wide variety of information which included the time each project member spent editing source files, code structural metrics, the occurrence of unit tests and their results, and system test coverage.

[1] http://hackystat.ics.hawaii.edu

6.2 Case Study Design and Strategies

The software engineering classroom was an environment where the use of software project telemetry could be mandated. Part of the course goal was to let the students gain hands-on experience with metrics collection and analysis. The software project telemetry system was used as a tool by the students to collect and analyze their product and process metrics.

Apart from test driving the software project telemetry system, the purpose of this case study was to collect the students' opinion about software project telemetry such as which part of technology worked, which part didn't, which part the students liked, which part they didn't like, and how the system could be improved.

The mixed methods paradigm was adopted in this case study. Since this was mainly a naturalistic study exploring how software project telemetry worked for software engineering students, priority was skewed toward qualitative information obtained during this case study. Quantitative data will be used to corroborate the qualitative findings.

The two most popular approaches in qualitative data collection are survey (either interview or questionnaire) and ethnographic observation. The advantage of ethnography was that it could enable me to observe how the students interacted with software project telemetry and how the technology facilitated them make project management and process improvement decisions. However, there were 25 students enrolled in the two classes. Obviously following those 25 students around and watching them developing software was an infeasible solution. As a result, I decided to use a survey. The interview method was not chosen because out of concern that the students might feel pressured to give more favorable opinions to bias the study results. Instead, an anonymous questionnaire survey method was my final choice. It had advantages such as ease of administration, rapid turn around in data collection.

The decision to use questionnaire survey, at the same time, implied that I had to rely on the students' self-reported opinions to evaluate software project telemetry. This threat was mitigated through data triangulation. All telemetry analyses invoked by the students were logged. If they ran the analyses on a regular basis, then I would put more confidence in their survey responses. On the other hand, if it turned out that they seldom invoked the telemetry analyses, then I had to discount their opinions.

6.3 Researcher's Role

The instructor of the two software engineering classes in which this case study was conducted was Dr. Philip Johnson. He is my dissertation adviser. The software project telemetry system is implemented by me. It was used as a tool by the students to collect and analyze their software product and process metrics. I helped the instructor predefine the telemetry charts and reports that we thought were most useful to the students. However, I did not participate in the teaching of the course.

6.4 Data Collection and Analysis Procedures

This case study collected data from 2 sources:

- A survey questionnaire was distributed at the end of the semester asking the students' opinion of software project telemetry.
- The software project telemetry system was instrumented. It logged all telemetry analysis invocation information.

The 2 sources of data were integrated at data interpretation phase, with priority skewed toward qualitative information and quantitative data corroborating the qualitative findings.

6.4.1 Survey Questionnaire

The survey was conducted through a written questionnaire administered on the last day of instruction. The questions covered metrics collection overhead, telemetry analysis usability and utility, as well as the students' perception whether software project telemetry was a reasonable approach to process improvement and project management in "real world" settings.

Each question was represented by a statement. For example, when collecting information about telemetry analysis utility, I made the statement *"telemetry analyses have shown me valuable insight into my and my team's software development process"*. I then asked the students to rank their feelings toward the statement on a scale from 1 to 5:

- 1 = strongly disagree
- 2 = disagree
- 3 = neutral
- 4 = agree
- 5 = strongly agree

The 6th option *“not applicable”* was provided as well to allow the students to skip the questions which they were unable to answer. At the end of each question, I provided large empty space to allow the students to record any related comments such as justification or elaboration of the answer.

The last page of the questionnaire was a free response section where the students were encouraged to supply any additional comments such as their general opinions toward software measurement, their concerns about the way software project telemetry handled their personal process data, their suggestions on how the system could be improved, etc.

The actual questionnaire is available in Appendix D for further reference.

6.4.2 System Invocation Log

The software project telemetry system exposes a web interface though which users can invoke telemetry analyses over his own or his team’s software product and process metrics. The system deployed for the classroom use was instrumented with automatic logging facility. Analysis invocation time, user name, and full request string are recorded.

6.5 Threats, Verification, and Validation

6.5.1 Measurement Validity

Measurement validity concerns about whether we are measuring what we are supposed to measure and whether the instrument can measure the data correctly.

The survey was conducted less than one week before the final examination. Students might be concerned that their response would influence their final grades and thus would comment more favorably toward software project telemetry. The threat was addressed in several different ways:

- First, the survey was anonymous and the students were specifically instructed not to reveal their names in their responses.
- Second, I personally assured the students that their responses would be sealed until after their instructor had turned in their final grades.
- Third, this entire research was conducted under the approval of the "committee for the protection of human subjects" at University of Hawaii. All research participants are protected by the committee.
- Lastly, all telemetry analysis invocations were logged. The students's analysis invocation information were used to assess "truthfulness" of their opinion (see Section 6.6.3).

6.5.2 Internal Validity

Internal validity is related to cause and effect, in other words, whether the treatment (using software project telemetry) actually causes the observed results (an improvement in software development process).

This case study did not employ ethnography to observe how software project telemetry impacted the students' software development process, because it was infeasible to track the activities of all the 25 students at the same time. Instead, a survey questionnaire was used. To mitigate this threat, I included a question asking the students whether they felt that their development actions had a causal relationship to telemetry streams (survey question 5). In other words, does telemetry provided a model for development that actually reflected changes in their behavior? This might not be enough to get a handle on causality, but this question will be further addressed in Chapter 7 where ethnography was employed to observer the impact of software project telemetry on CSDL developers' build process.

6.5.3 External Validity

External validity refers to the extent to which the results obtained in this case study can be generalized to a larger population.

This case study drew results from a limited sample size (25 students). All survey participants were computer science students taking either senior undergraduate or introductory graduate software engineering classes at the University of Hawaii in Spring 2005. The students might have a relatively homogeneous background in software development which might not be representative of software developers at a whole. The context of this study was a course setting. Course projects tend to be smaller and narrower in scope.

These external threats are limitations of this case study. The best way to address them is to conduct more case studies in different environments. Chapter 7 and 8 report on two more case studies in CSDL and Ikayzo respectively.

6.6 Results

All of the 25 students enrolled in the two software engineering classes participated in this study, of which 9 were from the senior undergraduate section and 16 were from the introductory graduate section. The students had fairly diversified background. Their total programming experience, as defined from the first “Hello World” toy application, ranged from 3 to 25 years, with a mean of 6.92 and a standard deviation of 4.43. Their paid professional experience[2] ranged from 0 to 8 years, with a mean of 1.27 and a standard deviation of 2.10 [3]. The survey was conducted in a normal class session (on the last day of instruction), and the response rate is 100%.

6.6.1 Results from Individual Question

The individual survey questions are listed below along with the results. Each question was in the form of a statement, and the students were asked to to circle the number that most closely matched their feelings about the statement. The options were 1 (strongly disagree), 2 (disagree), 3 (neutral),

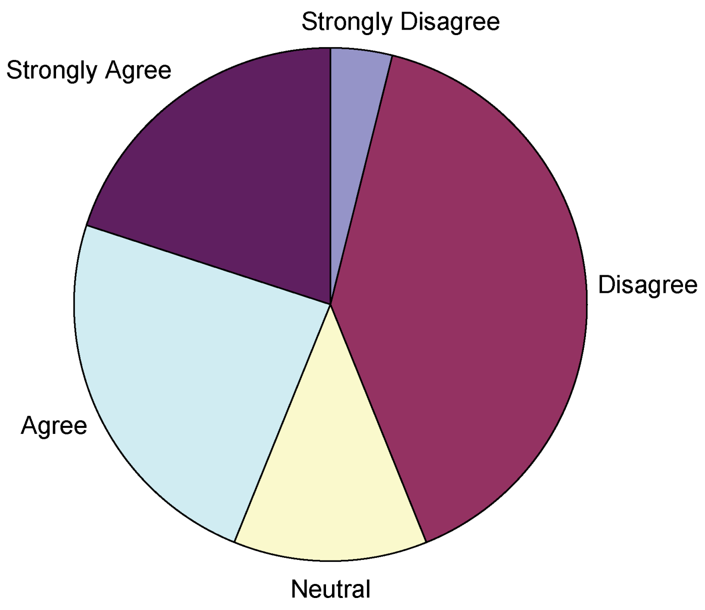
[2]In this question, I specifically asked the students to exclude the experience of half-time or less than half-time on-campus employment, such as student helper or research assistant, even if they were paid to program.

[3]One student did not answer this question and thus was not included in the statistics.

4 (agree), 5 (strongly agree), and 6 (not applicable). The resulting statistics were computed by excluding those "not applicable" answers.

Statement 1: I have no trouble installing and configuring the sensors.

Software project telemetry utilizes sensors to collect metrics, which intends to make the data collection process transparent and unobtrusive to developers. The sensors must be installed and configured properly before they can do their jobs. This question was designed to gather information about the one-time setup cost of the sensors.



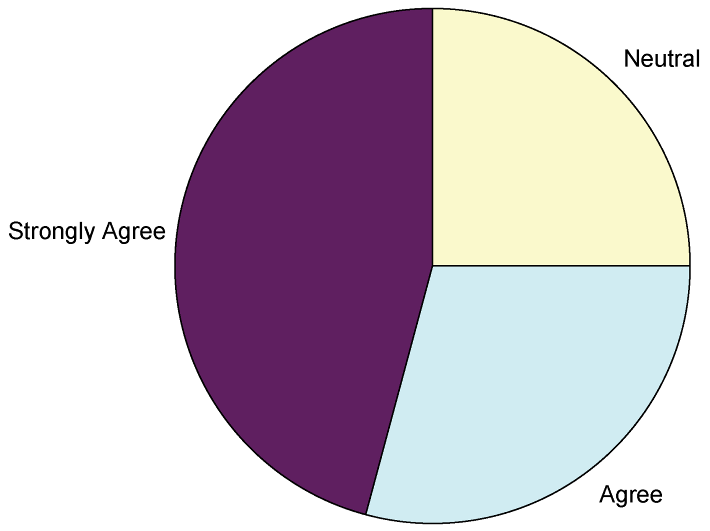
Response Rate: 25/25

Strongly Disagree	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
1	*10*	*3*	*6*	*5*

It turned out that installation and configuration of the sensors involved quite complex procedures. 40% of the respondents did not agree with the statement. One of the students wrote that he/she was still having troubles with the installation of some of the sensors at the end of the semester. Most students expressed the wish to have an all-in-one intelligent graphical user interface to install and configure the sensors.

Statement 2: After sensors are installed and configured, there is no overhead collecting metrics.

This question was designed to gather information about software metrics collection overhead as well. However, it had different goal than the previous question. This question focused on long-term chronic metrics collection overhead, while the previous one emphasized on the up-front sensor setup cost.



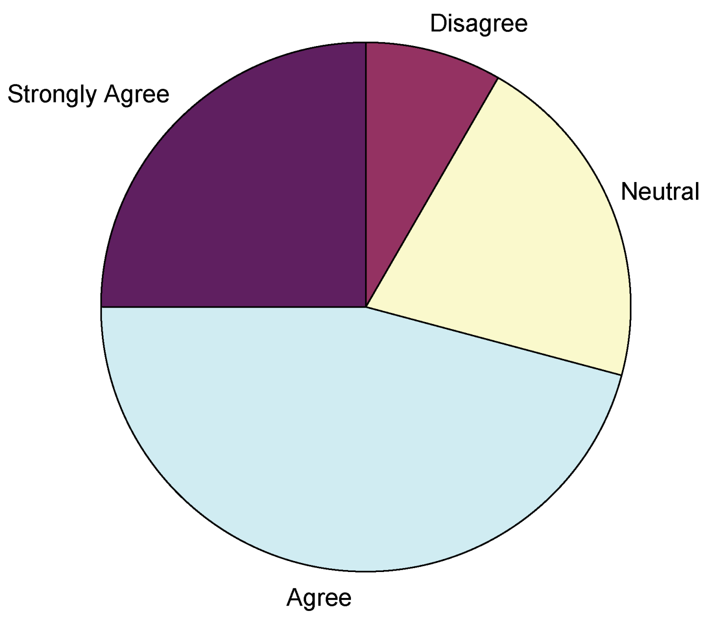
Response Rate: 24/25

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
0	*0*	*6*	*7*	*11*

The sensor-based metrics collection approach adopted in software project telemetry appeared to have achieved its design goal of eliminating long-term chronic data collection overhead. No one disagreed with the statement. The reason that some students chose "neutral" was mainly because they encountered bugs in one of the sensors released prematurely in order to meet the course schedule.

Statement 3: It's simple to invoke predefined telemetry chart and report analyses.

The software project telemetry implementation offered both an expert interface where users could use telemetry language to interact with the system and an express interface which allowed users to retrieve predefined telemetry charts and reports. The telemetry language was not introduced in the classes, nor was the expert interface. The instructor and I predefined dozens of telemetry charts and reports that we thought were most useful to the students. This question was intended to gather information about the usability of the express interface.



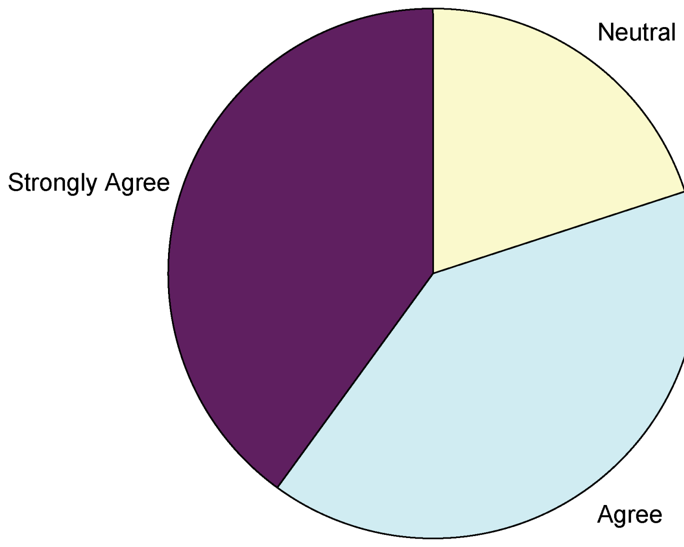
Response Rate: 24/25

Strongly Disagree	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
0	*2*	*5*	*11*	*6*

Though most students agreed that the express interface was easy to use, they thought it could be improved. It turned out that a major problem involved the input of telemetry analysis parameter values. Different telemetry charts or reports had different parameter requirements, but it was hard to tell from the user interface what parameters were expected.

Statement 4: Telemetry analyses have shown me valuable insight into my and / or my team's software development process.

One of the design goals of software project telemetry is to make the development process as transparent as possible so that software development problems can be detected early. This question was intended to measure whether software project telemetry had achieved that goal from the perspective of software developers.



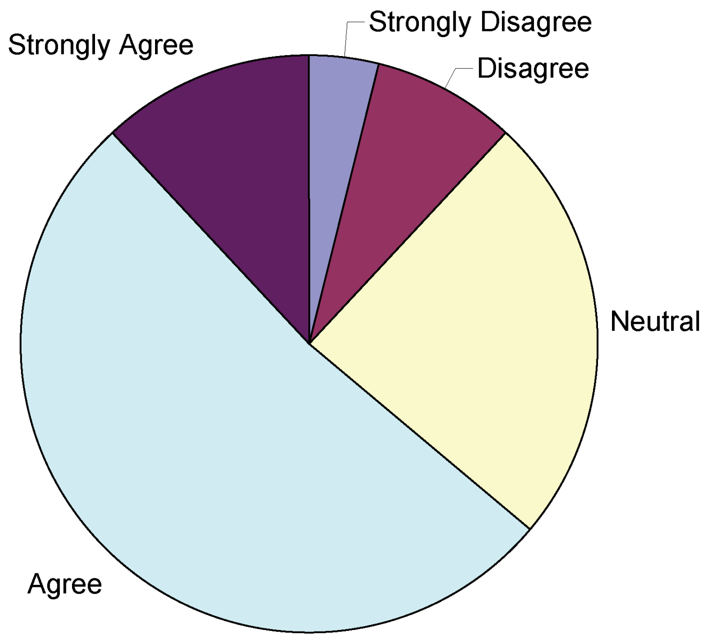
Response Rate: 25/25

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
0	*0*	*5*	*10*	*10*

No one disagreed with the statement. This indicated that software project telemetry had largely achieved the goal of making development process transparent. However, some students expressed concern about data privacy. Though steps have been taken during system design to only allow data sharing among project members, it seems that we have to do more work with data privacy.

Statement 5: Telemetry analyses have helped me improve my software development process.

Software project telemetry helps a developer improve his/her development process by making the development process transparent and the information available to the user, but it's not its design goal to make process improvement decisions on behalf of the user. Whether a developer can improve his/her development process depends crucially on (1) his/her ability to interpret telemetry results and take actions accordingly, and (2) whether telemetry analysis is able to deliver the relevant information in an easy-to-understand way. This question was intended to ask the students whether there was any self-perceived process improvement after using the system.



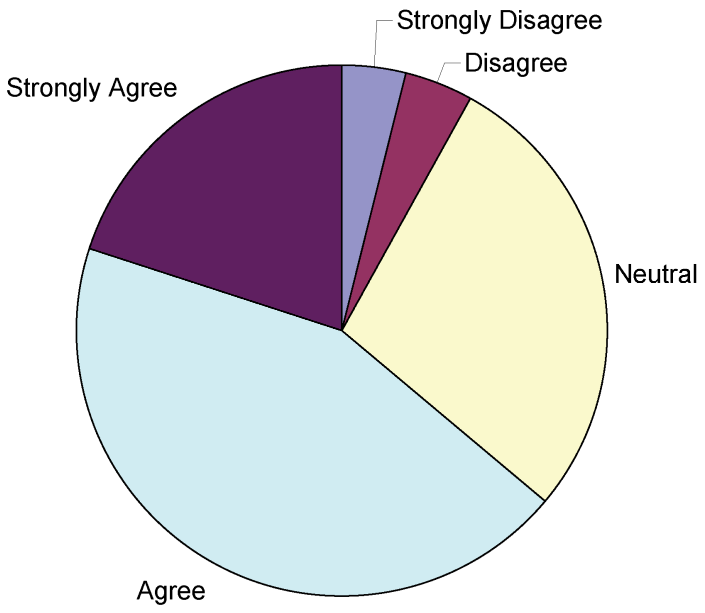
Response Rate: 25/25

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	*2*	*6*	*13*	*3*

Most students concurred with the statement that telemetry analyses had helped them improve their software development process. However, due to the limitation of this study, questions remained. I was not able to tell whether the students' self-perceived improvement in their development process was due to the use of the software project telemetry system, or the fact that they learned new development best practice in class, or both.

Statement 6: If I was a professional software developer, I will want to use telemetry analyses in my development projects.

This question was intended to ask the students whether they perceived software project telemetry as a reasonable approach to process improvement in “real” software development settings from the perspective of a developer.



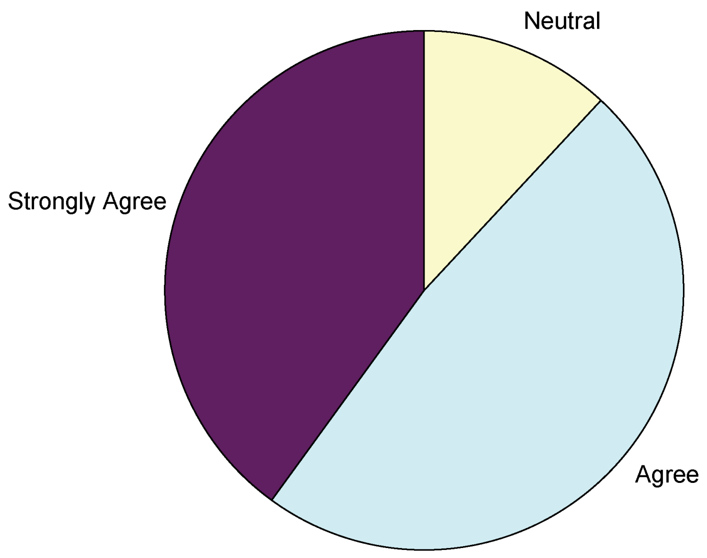
Response Rate: 25/25

Strongly Disagree	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
1	*1*	*7*	*11*	*5*

The majority of the students confirmed the value of software project telemetry as a reasonable approach to software process improvement. However, some of them expressed the concern about sharing private personal process data with others. For example, one student said *“I don’t want to show the data to my boss.”*. Data privacy is a serious issue. If it can not be handled properly, it might be a significant adoption barrier to the technology.

Statement 7: If I was a project manager, I will want to use telemetry analyses in my development projects.

Software project telemetry can be used by a project manager to monitor development progress. This question was intended to ask the students whether they perceived software project telemetry as a reasonable approach to project management in “real” software development settings from the perspective of a project manager.



Response Rate: 25/25

Strongly Disagree	**Disagree**	**Neutral**	**Agree**	**Strongly Agree**
0	*0*	*3*	*12*	*10*

Nobody disagreed with this statement. When used as a management tool, there is no data privacy issue from the perspective of a project manager. The more information that is available to the project manager, the better informed decision he/she can make.

6.6.2 Results from Free Response Section

The students provided a lot of textual feedback. From the feedback, several themes were identified. The identified themes are listed in bold face followed by the students' actual comments.

- **Having an automated and unobtrusive mechanism for metrics collection is helpful.**

  > *"Overall it is an incredible tool that generally makes software development metric collection effortless."*

- **Software project telemetry offers powerful information and insights into the development process.**

  > *"There is powerful information and insights to gain."*

  > *"The first key is that analyses have made me aware."*

  > *"(Telemetry analyses have helped me improve my software development process) compared to what I learned in class, from students, and on the web."*

  > *"(Telemetry analysis) makes me more aware of what others are doing – good or bad?"*

- **Understanding and interpreting telemetry data is the key to get the value from the tool. This requires commitment to metrics-based process improvement.**

  > *"I would say that understanding and interpreting the results to benefit the development process is the key ingredient to getting value from the data. Does a team of software developers understand the domain to make use of the information?"*

  > *"Have not done enough projects to get a pattern."*

- **Sensor installation and configuration are too complex.**

  > *"It took some time to install it."*

  > *"I thought the instructions were much too detailed. I got lost with details."*

  > *"No all-in-one installer. Too much manual work."*

  > *"I could not figure out that I need to put sensor.properties to .hackystat folder."*

  > *"I still have problems with some sensors (at the end of the semester)."*

“Please make the installation process easier.”

“(It) should have some mechanism to switch on the sensors easily instead of going into the sensor property file to change the true to false.”

“(You) should really consider creating installer scripts.”

“(They) need GUI driver process.”

- **Some sensors do not seem working correctly.**

  This is a complex issue. There are several reasons that could cause sensors seemingly not working as expected: (1)programming bugs in sensor code, (2)incorrect sensor configurations or project settings, or (3) inappropriate interpretation of metrics data.

  “The sensors of the Jupiter does not work correctly sometimes, and did not sense any data and send it to the server.”

  “I had a lot of problems getting Jblanket to work on a web page with http unit. For some version, it kept closing the web application from running on the server, even with no dependencies.”

  “I spend 2 hours one night to debug a problem with the Ant build file and end up with only 15 minutes on Hackystat.”

- **The web interface of telemetry analysis could be more user-friendly.**

  “The website interface is really improvable.”

  “I think the Hackystat server website can be more user friendly. It took me a while to get used to the page and find the relevant (telemetry) charts I wanted.”

  “The user interface is a little bit confusing. It’s hard to click on Extras (link) when there is no information about what Extra (link) does.”

  “Interface to Hackystat website (for telemetry analysis) yields too many options on pages. Could use a simplified design.”

  “I think the way a developer views a (telemetry) report needs to be simplified. Of course, I can see some people would want to customize their own (telemetry) reports.”

  “Some (telemetry analysis invocation) require unknown parameters.”

  “Others (telemetry analysis invocations) require parameters, but no instructions on what those parameters might be.”

> *"They (telemetry analysis invocations) don't work so well due to the last parameter option. What goes there? How about a help link for each option?"*
>
> *"The (telemetry) report names aren't that descriptive, and the the parameters needed were confusing."*

- **Developers have privacy concerns with their personal process data.**

  > *"(Software project telemetry data are) good if used correctly."*
  >
  > *"(Telemetry analyses) makes me more aware of what others are doing – good or bad?"*
  >
  > *"I don't want to show the data to my boss."*
  >
  > *"Maybe (you should) consider more information gathering to give programmers insights v.s. giving manager insight."*

- **Software project telemetry is better suited as a project management tool.**

  > *"(It is) more useful for management."*
  >
  > *"I want to know what people are doing."*

- **Miscellaneous Comments**

  > *"Eclipse sensor updates too often. Ever time when I start Eclipse I had to download new version. It was too much overhead for me. But I do not want to disable automatic update, because I will forget to update if it is disabled. Can you limit the sensor updates to once a week or twice a month?"*
  >
  > *"I'll be concentrating on my work. Stats don't really matter."*
  >
  > *"(Telemetry analysis is useful) only if I was in a development team. If I was alone I am not sure I'd use it."*

6.6.3 Results from Telemetry Analysis Invocation Log

The telemetry analysis invocation log was kept in space-delimited text files. I have just finished a program that imports the log records into a database, but it takes some time to analyze the data.

I will plot each student's invocation count on a daily or weekly bar chart. The information gives indication of the overall popularity of the telemetry mechanism. It could be used to triangulate the previous survey findings. There are three possible scenarios:

- The survey results revealed that most of the students though software project telemetry was a useful approach to software project management and process improvement in general, though certain aspects of it could be improved. I will have more confidence in the survey results if the telemetry analysis invocation log indicates that most of them invoked the telemetry mechanism a lot, especially on a regular basis.

- The survey did not reveal any overly negative comments about software project telemetry. If the log indicates that most of the students did not invoke the telemetry mechanism very much or they stopped using it after an initial period, then I will have to discount the survey results. Perhaps the inconsistency is caused by the fact that the students were concerned that negative comments would bring negative grades.

- The survey was anonymous, which means that there was not enough information to connect individual telemetry analysis invocation to individual survey response. If the log indicates that some of the students invoked telemetry mechanism a lot while others did not use it very much or stopped using it after a while, then I will not be able to use the data to triangulate the survey findings.

Once the analysis of invocation log is complete, I will know which scenario I am in.

6.7 Conclusion

The case study yielded valuable insights into software project telemetry as well as the current implementation of the system.

An automated and unobtrusive metrics collection mechanism is crucial to the success of a metrics program. From the student's feedback, sensor-based metrics collection approach appears to have eliminated long-term data collection overhead successfully. However, the one-time setup cost of the current sensors is still too high. While this is not a major issue in a setting where the use of the technology is mandatory, it could cause adoption barrier in other environment. Many survey participants have expressed the wish to have an all-in-one intelligent graphical user interface to install and configure the sensors. Fortunately, such an installer is now available.

Software project telemetry provides powerful insights by making the software development process transparent. Participants in this study generally agreed that they were made more aware of both their own and their team's development process as a result of using the system. An important factor to benefit from software project telemetry is the understanding and appropriate interpretation of telemetry metrics. For example, there were several reports during the semester that the sensors did not seem to collect metrics correctly, or the analyses did not seem to compute the data as expected. Some were caused by inappropriate interpretation of the results. It seemed that effort-related metrics were most susceptible to mis-interpretation. As far as the implementation is concerned, many participants voiced that the current telemetry analysis interface worked but could be more user-friendly.

There was a data privacy issue, especially with effort-related metrics data. Some students concerned that their personal process data might be misused. We were very well aware of the issue while designing the system, and had taken steps to limit the scope that the data could be accessed. However, it seemed hard to reconcile the difference between project management metrics requirements and perfect personal privacy protection. Some participants expressed that they would not want to share personal metrics with others, while other participants said they would like to know what other people were doing. As with all successful metrics programs, correct interpretation and proper use of metric data are crucial, as well as developer understanding, trust, and support.

Chapter 7

Case Study in CSDL

This chapter describes a case study that will be conducted in Collaborative Software Development Lab at University of Hawaii, where a large scale software project (100 KSLOC) is being developed and maintained. The project has experienced a significant rate of integration build failure (88 failures out of 300 builds in 2004). The case study has dual objectives:

- To use software project telemetry to understand and improve CSDL build process.
- To evaluate software project telemetry at the same time in CSDL (an environment typical of traditional software development with close collaboration and centralized decision-making).

This chapter begins with a description of CSDL environment and its build problem in Section 7.1. Section 7.2 introduces case study participants and the researcher's role. Section 7.3 outlines the design and strategies of this case study and offers rationale for the decisions. Section 7.4 describes data collection and analysis procedures. Section 7.5 discuses the limitations and threats. The expected results are described in Section 7.6.

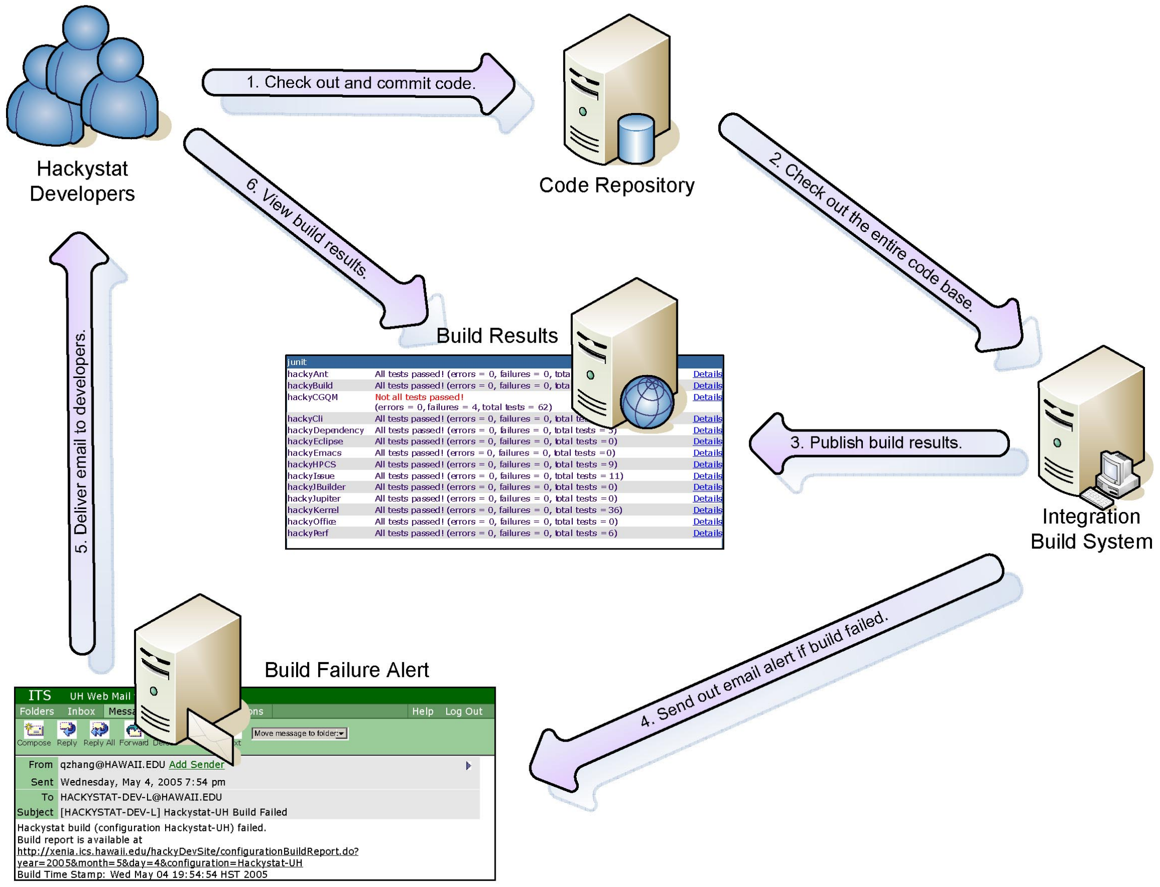
7.1 CSDL Setting

The Collaborative Software Development Laboratory (CSDL) is a software engineering research lab at University of Hawaii. Its mission is to provide a physical, organizational, technological, and intellectual environment conductive to collaborative development of software engineering skills. Hackystat is a software system developed and maintained by CSDL. Its aim is to provide a common framework for automated metrics collection and software engineering experimentation.

7.1.1 Build Process

Hackystat consists of over 100 thousand lines of code at the time of this writing. The code is organized into over 30 different modules, with 5 – 10 active developers. The sources are stored in a central version control repository, which supports concurrent development by allowing developers to checkout the latest version of the sources and commit their changes. Developers rarely compile, build, and test the system against the entire code base. Instead, they often work with a subset of the modules relevant to their assignments.

In order to handle full system build and test, I designed and implemented an automated integration build tool, which checks out the latest version of the entire code base, compiles, builds, and tests the system. The build tool is designed so that it starts the integration build automatically at night if it detects any change in the code repository in the previous day. If there is any build error, an email is sent to the development team. Figure 7.1 is a graphical illustration of the build process.


Hackystat Developers
1. Check out and commit code.
Code Repository
2. Check out the entire code base.
Integration Build System
3. Publish build results.
Build Results
4. Send out email alert if build failed.
Build Failure Alert
5. Deliver email to developers.
6. View build results.
ITS UH Web Mail
Folders Inbox Messa ons Help Log Out
Compose Reply Reply All Forward
Move message to folder:

junit		
hackyAnt	All tests passed! (errors = 0, failures = 0, tota	Details
hackyBuild	All tests passed! (errors = 0, failures = 0, tota	Details
hackyCGQM	Not all tests passed! (errors = 0, failures = 4, total tests = 62)	Details
hackyCli	All tests passed! (errors = 0, failures = 0, total te	Details
hackyDependency	All tests passed! (errors = 0, failures = 0, total tests = 5)	Details
hackyEclipse	All tests passed! (errors = 0, failures = 0, total tests = 0)	Details
hackyEmacs	All tests passed! (errors = 0, failures = 0, total tests = 0)	Details
hackyHPCS	All tests passed! (errors = 0, failures = 0, total tests = 9)	Details
hackyIssue	All tests passed! (errors = 0, failures = 0, total tests = 11)	Details
hackyJBuilder	All tests passed! (errors = 0, failures = 0, total tests = 0)	Details
hackyJupiter	All tests passed! (errors = 0, failures = 0, total tests = 0)	Details
hackyKernel	All tests passed! (errors = 0, failures = 0, total tests = 36)	Details
hackyOffice	All tests passed! (errors = 0, failures = 0, total tests = 0)	Details
hackyPerf	All tests passed! (errors = 0, failures = 0, total tests = 6)	Details

From	qzhang@HAWAII.EDU Add Sender
Sent	Wednesday, May 4, 2005 7:54 pm
To	HACKYSTAT-DEV-L@HAWAII.EDU
Subject	[HACKYSTAT-DEV-L] Hackystat-UH Build Failed

Hackystat build (configuration Hackystat-UH) failed.
Build report is available at
http://xenia.ics.hawaii.edu/hackyDevSite/configurationBuildReport.do?
year=2005&month=5&day=4&configuration=Hackystat-UH
Build Time Stamp: Wed May 04 19:54:54 HST 2005

Figure 7.1. Hackystat Build Process

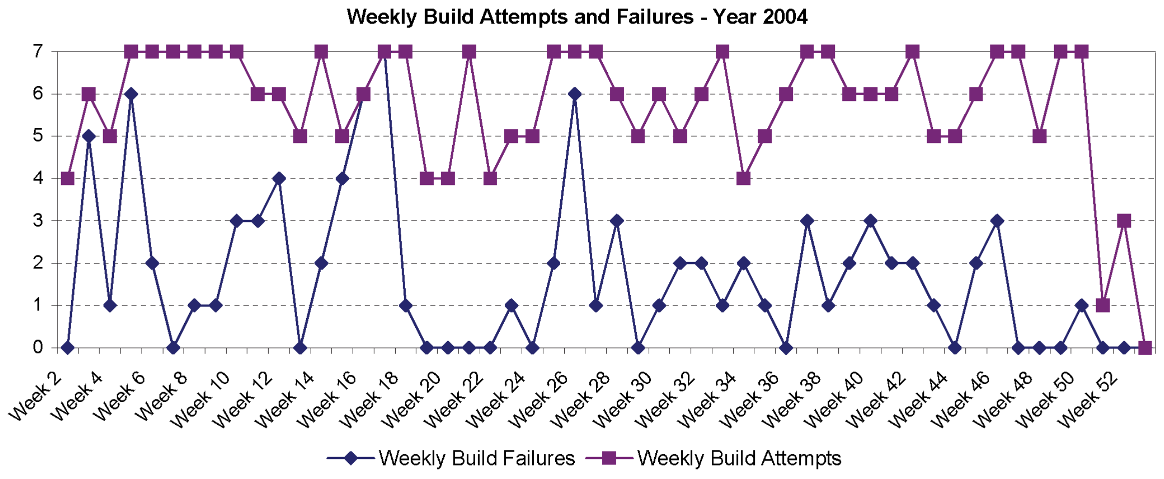


Figure 7.2. Weekly Hackystat Build Failures and Attempts for the Year 2004

7.1.2 Build Problem

The automated build tool was deployed in CSDL in the summer of 2003. At the end of 2004, I conducted a study on the build results and found that integration build failure rate was significant. There were 300 days in which Hackystat code repository was changed for the entire year of 2004. As a result, there were 300 integration build attempts by the automated build tool. Out of those 300 build attempts, the build failed 88 times with a total of 95 errors.

Figure 7.2 shows the build failure information on a weekly basis. The top line indicates the number of build attempts in each week. Since the automated build tool initiates an integration build only when it detects source code change in the previous day, the maximum number of build attempts in a week is seven. The bottom lines represents the number of build failures in each week. A few weeks saw 100% build failure (e.g. week 16 – 17), while a few other weeks saw 100% build success (e.g. week 19 – 22). In most weeks, the build failure rate fluctuated between 20% to 50%. The overall average failure rate was 29.33%.

The loss of productivity due to the integration build failures was substantial, since each build failure generally required one or more developers to stop concurrent development, diagnose the problem, and determine who was responsible for fixing the error. Often times, other developers had to wait until the corrections were made before they could check out or commit additional code.

	Coding Standard Violation	**Compilation Failure**	**Unit Test Failure**	**Other Errors**
Count	*14*	*25*	*40*	*16*
Percentage	*14.74%*	*26.32%*	*42.11%*	*16.84%*

Table 7.1. CSDL 2004 Integration Build Failures

7.1.3 Initial Investigation

Several metrics on different aspects of the development process were used in the initial investigation:

- *Active Time* — The proxy for developer effort writing or modifying code inside an IDE.
- *Commit* — The commit of a file to a source code repository.
- *Build* — The invocation of a build tool (Apache Ant in this case) and the build result.
- *Churn* — The lines added and deleted in the source during some period of time.

The investigation revealed several interesting findings:

- The 95 integration build failures in 2004 could be partitioned into 4 major categories as shown in Table 7.1: coding standard violation (14), compilation failure (25), unit test failure (40), and others (16). The 16 “other errors” consisted of 8 build script errors, 3 platform errors, and 5 unknown errors. Platform errors refer to the ones caused by external factors, such as network unavailability and operating system failure, which is generally beyond the control of the developers. The unknown errors are the ones that I was unable to assign cause because of insufficient archived information.
- The two Hackystat modules with the most dependencies to other modules had the two highest numbers of build failures, and together they accounted for 30% of the total failures.
- The days with build failures had a statistically significant greater number of distinct module commits than the days without build failures.
- There was no statistical correlation between integration build failure and the number of lines of code committed or the amount of active time spent before the commit. In other words,

whether you worked 5 hours or 5 minutes, or whether you changed 5 lines of code or 500 did not change the odds of causing an integration build failure.

- There were substantial differences (5 – 10 fold) between experienced and new developers with respect to their propensity to fail the integration build.
- Most integration build failures were confined within single Hackystat module. 74% of the failures could have been prevented if the developer could test the modules they were working on before committing the changes to the central code repository. A further 6% could have been avoided if the developer could run a full system test over all Hackystat modules before committing their changes.

7.1.4 Initial Hypothesis

Perhaps the most significant finding was that most integration build failures were confined to a single module, and that 74% – 82% of them were actually preventable if the developers could build and test the system on their workstation before committing the changes to the central code repository. Therefore, the simplest process improvement recommendation would be to require all developers to test their modules every time before making a commit. The problem with this recommendation was the large amount of overhead involved. A system test takes at least 10 to 20 minutes depending on the number of modules and the speed of the workstation. Though some commits without testing caused integration build failures, many did not. The cost of productivity loss associated with this naive process change might well exceed the benefit it would bring.

The initial analysis also suggested that the causes of the integration build failures were quite complex. They involved developer's familiarity with the system, the actual changes made to the code, the dependency relationships among the modules, etc. However, there was no statistical correlation between build failure and the number of files committed or developer active time. It would be difficult to adopt the traditional approach to build an analytical model to predict the probability of integration build failure in order to forewarn the developers.

The approach to improve CSDL build process in this case study came from the recognition of the following trade-off:

- Reducing the number of integration build failures could increase developer productivity.

- Performing local system test could reduce the number of integration build failures at the expense of developer overhead.

As a result, the goal of build process improvement is not to minimize the absolute number of integration build failures. Instead, the goal is to increase the overall developer productivity by making their software development process transparent, providing them with process feedback, and helping them understand plausible causes of integration build failures, so that they could make an informed decision when a local system test would likely to be necessary before committing the changes.

7.2 Study Participants and Researcher's Role

I joined CSDL in 2003. Currently CSDL has 6 members. Three of them are Ph.D. students (including me). They are hired as research assistants (0.50 full time equivalence) by Dr. Philip Johnson to develop and maintain the Hackystat framework. At the same time, they are doing their own doctoral research using the framework as a software engineering experimentation tool. The other 3 members are undergraduate students in their final semester working on Hackystat for course credits. They are top students from software engineering classes.

Dr. Philip Johnson is the director of CSDL. He is my dissertation adviser. He is the project manager of Hackystat responsible for major architectural design and scheduling decisions. He also contributes to Hackystat development.

The software project telemetry system used in this case study is designed and implemented by me as part of my dissertation research. It is built on top of the Hackystat foundation, leveraging its generic framework of metrics collection and project management.

The 5 CSDL members (excluding me) and Dr. Philip Johnson (as project lead and developer) are the participants of this case study. Though I am also a member of CSDL and contribute to Hackystat development, I will exclude my development activities from this case study.

7.3 Case Study Design and Strategies

The purpose of this case study is two-fold:

- To use software project telemetry to understand and improve CSDL build process.
- To evaluate software project telemetry during the process.

The second objective is more important than the first one. I want to understand how software project telemetry technology can be applied in a more traditional software development environment such as CSDL with relatively centralized decision-making and close collaboration. I want to know how the developers make use of telemetry information to make process improvement decisions. I want to know their opinions about the technology. I want to find out what works, what doesn't work, and what can be improved.

The organization of this section is as follows. Section 7.3.1 elaborates on the steps to improve CSDL build process with software project telemetry. Section 7.3.2 details evaluation strategies.

7.3.1 Improving CSDL Build Process

In order to improve CSDL build process, I will use software project telemetry to collect and analyze relevant process and product metrics, and make the information transparent to the entire development team. I will provide process feedback through both pull style telemetry analysis and push style telemetry alert. The hypothesis is that when the developers are made more aware of their development process and night integration build results, they can become better at deciding when a local system test is likely to be necessary before committing their changes.

The 2004 CSDL build analysis was performed manually which required a tremendous amount of effort. The only information available was the archived email alerts notifying the developers of integration build failures. As a result, I had to checkout the snapshot of Hackystat source code for each day in 2004, simulate the build, and find out where and why the build failed. I spent the entire winter break (1 month) to perform the analysis.

The manual approach will not work for CSDL members to improve their build process, simply because of the tremendous amount of overhead associated with manual metrics collection and analysis. Fortunately, we can use software project telemetry to automate these time-consuming metrics collection and analysis tasks. As the first step of this case study, I developed a build sensor, which runs on both the integration build server and individual developer's workstation. The sensor collects the following information automatically and unobtrusively: build time, build context (environment information about the build, such as who starts the build, on which computer the build is started, which modules are being built, build target, etc.), and build results (in case of build failure, which module, which file, and which step is the cause of the failure). The build data, combined with other metrics collected by the existing sensors, such as IDE sensor (who is editing which file), CVS sensor (who has committed which file), provide a holistic picture of the entire team's build process.

Pull Style Telemetry Analysis

The software project telemetry system, which is first introduced in Chapter 4, will be used in this case study. The implementation contains generic support for telemetry-style software metric analysis such as telemetry definition management and telemetry language interpreter. I will customize the system with an initial set of telemetry streams, based on the analysis of 2004 CSDL integration build metrics, to provide process feedback to CSDL developers. This set may change during this case study as I understand more about the role of software metrics in improving CSDL build process.

The initial set of telemetry streams I plan to implement can be divided into the following categories:

1. *Telemetry streams related to integration build failures — team level*

   (a) Integration build failure count, with breakdown to failure types such as coding standard violation, compilation failure, and unit test failure.

   (b) Integration build failure count, with breakdown to modules in which the failure occurs.

   The purpose of these telemetry streams is to inform the developer of the characteristics of build failures such as its distribution over failure type or Hackystat modules, and detect any change in build failure patterns over time.

2. *Telemetry streams related to local system test and commit — individual level*

(a) Source code commit batch count, with breakdown to commit types (with respect to local system test).

(b) Local system test count, with breakdown to test target such as source code format check, system compilation attempt, and unit test attempt.

(c) Local system test count, with breakdown to test result (success or failure).

(d) Time spent on local system test.

(e) Integration build failure count caused by insufficient local test.

A significant discovery while analyzing 2004 CSDL build failure metrics was that 74% to 82% of them were preventable if the developers could run test over the modules they were working on before committing their changes to the central code repository. As a result, the interaction between the developers' test and commit behavior is an important phenomena to be explored.

Source code commits are partitioned into 4 types with respect to local system test in this case study:

- Commit activity preceded immediately by a full set of successful local system test.
- Commit activity preceded immediately by a partial set of successful local system test.
- Commit activity preceded immediately by a failed local system test.
- Commit activity not preceded immediately by a local test (e.g. editing activity are detected).

It is not the design purpose of these telemetry streams to force the developers to run a local system test every time they wish to commit something. Instead, I want to give the developers insight about the relationship between their code commit pattern, local system test pattern, and integration build results, so that they can make trade-off decisions to increase overall development efficiency.

3. *Telemetry streams related to other relevant information — individual level*

(a) Number of files committed, and number of modules committed.

(b) Developer active time.

Push Style Telemetry Alert

The current integration build failure email alert contains only a link to the build report. While build reports are excellent source to figure out where the build fails, it does not show process level information. In other words, it helps the developers fix bugs, but does not help them improve their build process.

I will augment the build failure email alert with process level information. It will contain two links: (1) the original link to the build report, and (2) a new link to the build process telemetry analyses. The email will also include summary information about each developer's commit and local system test behavior. More importantly, I will implement a mechanism that will automatically deduce the most likely individual developer who is responsible for integration build failure from telemetry data. Though there would be cases that the mechanism could give erroneous results, the idea is that when the analysis could point fingers to a particular developer, people would have more interest in the analysis and more incentive to avoid build failures.

The summary information will look like:

```
The following alert(s) were triggered today:

* CSDL Build Analysis Case Study:

  Integration build on 08-Oct-2005
  for hacky2004-all project FAILED.
  [hackyBuild] [JUnit Failure]
  [Plausible culprit: Developer B]

  Individual Developer Commit and Test Behavior:
    Total Batches of Commit
    [FullyTested / PartiallyTested / UnTested / FailedTest]

    Developer A:  1 [0 / 0 / 1 / 0]
    Developer B:  1 [0 / 0 / 0 / 1]
    Developer C:  1 [0 / 1 / 0 / 0]
    Developer D:  2 [0 / 0 / 2 / 0]
```

7.3.2 Evaluation of Software Project Telemetry in CSDL

In order to assess the effectiveness of software project telemetry, I will adopt mixed methods paradigm in this case study collecting and analyzing both qualitative and quantitative information. However, the priority will be skewed toward qualitative information with quantitative data corroborating qualitative findings.

There are mainly 2 choices to set up the assessment of software project telemetry in CSDL: either a rigid formal experiment or a naturalistic study. I choose naturalistic study over formal experiment in this case study.

The main advantage of a formal experiment is that controlling of compounding effects leads to maximization of internal validity and that there are relatively straight-forward statistical procedures to determine whether the use of software project telemetry is the cause of build process improvement. However, the reason why the experiment methodology is not chosen is as follows. First, CSDL has 6 developers at the time of this writing, which makes randomized selection of study participants or utilization of control groups practically infeasible. Second, three developers left and three new developers joined the group in the summer of 2005, which makes 2004 integrations build results and process metrics not directly comparable to those that will be obtained in this case study. Third, different developers take responsibility for different modules of Hackystat system with some module inherently more difficult to handle than others. Furthermore, tasks assigned to each developer change over time. These two facts make it hard to compare the process impact between different developers and to control for maturation effects. Perhaps, most importantly, software project telemetry is a young technology, and as such the more pressing need is to explore how the technology can be applied in some software development environment and find out what works and what needs to be improved. Therefore, it is more advantageous to adopt naturalistic approach to provide detailed description of the experience of the application of software project telemetry technology in CSDL, and to use quantitative information to corroborate the findings in qualitative analysis.

Qualitative data will be collected through ethnographic observations, personal diary, and developer interviews; while quantitative data will include integration build results, telemetry analysis invocation records, and individual developer's process metrics. The next section (Section 7.4) elaborates on data collection and analysis procedures.

7.4 Data Collection and Analysis Procedures

This case study will collect both qualitative and quantitative data from multiple sources. Both types of data will be collected concurrently.

7.4.1 Qualitative Data Collection

Qualitative data will be collected through ethnographic observations, personal diary, and developer interviews.

Ethnographic Observations

I will employ ethnography to record the observation of the developers' daily interaction with software project telemetry system and how they make process improvement decisions based on telemetry information. The record of observation will also include contextual information such as the tasks each developer is assigned to, the observed change in the build process, and their comment about software project telemetry.

Personal Diary

I will keep a personal diary of my interpretation of the observed facts such as daily integration build results, plausible causes of build failures, telemetry analysis results, and the change in each developer's build process.

Developer Interviews

I will interview the developers on a weekly or bi-weekly basis to find out the impact of software project telemetry on their build process and obtain their comments about the technology. The interview will be carried out in the following steps:

- Review past integration build results. Identify the cause for each build failure.

- Determine whether each build failure is avoidable or not. Find out whether the responsible developer has performed local test or not. If not, ask the developer whether it is due to neglect, overconfidence, or some other unavoidable reasons.
- Go over telemetry analysis results together with the developers. Identify the changes in their process.
- Reconcile my own interpretation with the developers' interpretation.
- Get developers' opinion about existing telemetry analysis.
- Ask developers what can be done to improve telemetry analysis and what other information might be useful.

7.4.2 Quantitative Data Collection

Quantitative data will be collected from the following sources.

Integration Build Results

The data is collected by the build sensor introducted in Section 7.3.1. The sensor is attached to the integration build server collecting build results automatically.

Individual developer's process metrics

They are the exactly the same metrics providing process feedback to the developers. These metrics capture information about source code edit, commit, local test, etc. They will be used to assess changes in individual developer's build process.

Telemetry Analysis Invocation Records

The software project telemetry system used in this case study will be instrumented with automatic logging facility. It will record all telemetry analyses invoked by the developers.

7.4.3 Data Analysis

Qualitative and quantitative information will be integrated at data interpretation phase with priority given to qualitative information. Quantitative data will be used to corroborate qualitative findings. I will either note the convergence of the findings as a way to strengthen the knowledge claims of the study or explain any lack of convergence that may result.

7.5 Threats, Verification, and Validation

7.5.1 Measurement Validity

This case study will draw conclusion from both qualitative and quantitative data. Quantitative data are collected automatically either by sensors or through system instrumentation. They all have pretty standard definition, and the risk of measurement validity is very low.

On the other hand, qualitative data used in this case study may suffer from measurement validity. The threat comes from the fact that I am a member of CSDL. Other members might be compelled to comment more favorably about software project telemetry. But there are several mitigating forces:

- I am no stranger to other CSDL members, and as a result my ethnographic observation will cause less disturbance on their normal activities.
- I have detailed knowledge about the project and have a good understanding about the nature of each developer's task assignment, which gives me great insight into their development process.
- Data from multiple source will be cross-validated to assess the effectiveness software project telemetry.

7.5.2 Internal Validity

Internal validity is related to causality: whether the introduction of software project telemetry in the CSDL development environment does indeed cause the improvement in the build process.

Though this case study adopts a naturalistic approach in which compounding factors such as maturation are not controlled, the risk of internal validity threat is mitigated through data triangulation. Data will be collected from multiple sources such as ethnography, personal diary, interviews, process metrics, and server instrumentation. All information will be reconciled and cross-validated at data analysis stage.

7.5.3 External Validity

CSDL environment resembles traditional software development environment in many aspects. A version control system stores the source code; an issue management system tracks features requirements, bugs, assigns tasks to developers; an integration system builds and tests the project automatically; and code reviews are conducted on a regular basis. However, it is still an academic environment and my have different constraints in software development than industrial environment.

The primary strategy to ensure external validity will be the provision of rich, thick, detailed descriptions of this case study. This way, anyone interested in extending CSDL experience will have a solid framework for comparison.

7.6 Expected Results

The use of software project telemetry will have impact on CSDL build process, though it is hard to tell to what extent the development process can be improved at this stage. However, regardless of the outcome, I will gain experience by applying software project telemetry to solve development process problems. This experience will enhance our understanding of the technology.

Chapter 8

Case Study at Ikayzo

This chapter outlines a case study of software project telemetry that will be performed at Ikayzo[1] with open-source project developers. Open-source project development environment is different from those in classroom or CSDL. The development is collaborated by geographically dispersed developers and project decisions are decentralized. It presents significant challenge for software project telemetry technology to be adopted.

This chapter begins with a description of Ikayzo in Section 8.1. Section 8.2 outlines the design and strategies of this case study and offers rationale for the decisions. Section 8.3 introduces data collection and analysis procedures. Section 8.4 describes researcher's role. Section 8.5 discusses the limitations and threats. The expected results are described in Section 8.6.

8.1 Ikayzo Setting

Ikayzo provides open-source projects with free hosting service, which includes source code version control, issue tracking and management, automated metrics collection and analysis, and wiki discussion board. All projects have to go through a screening process before they are hosted. Beyond that, Ikayzo does not participate in or have influence on the decision-making process of the hosted projects.

The metrics collection and analysis support is provided through the software project telemetry system. Unlike the case in the two previous studies where the developers have to install sensors

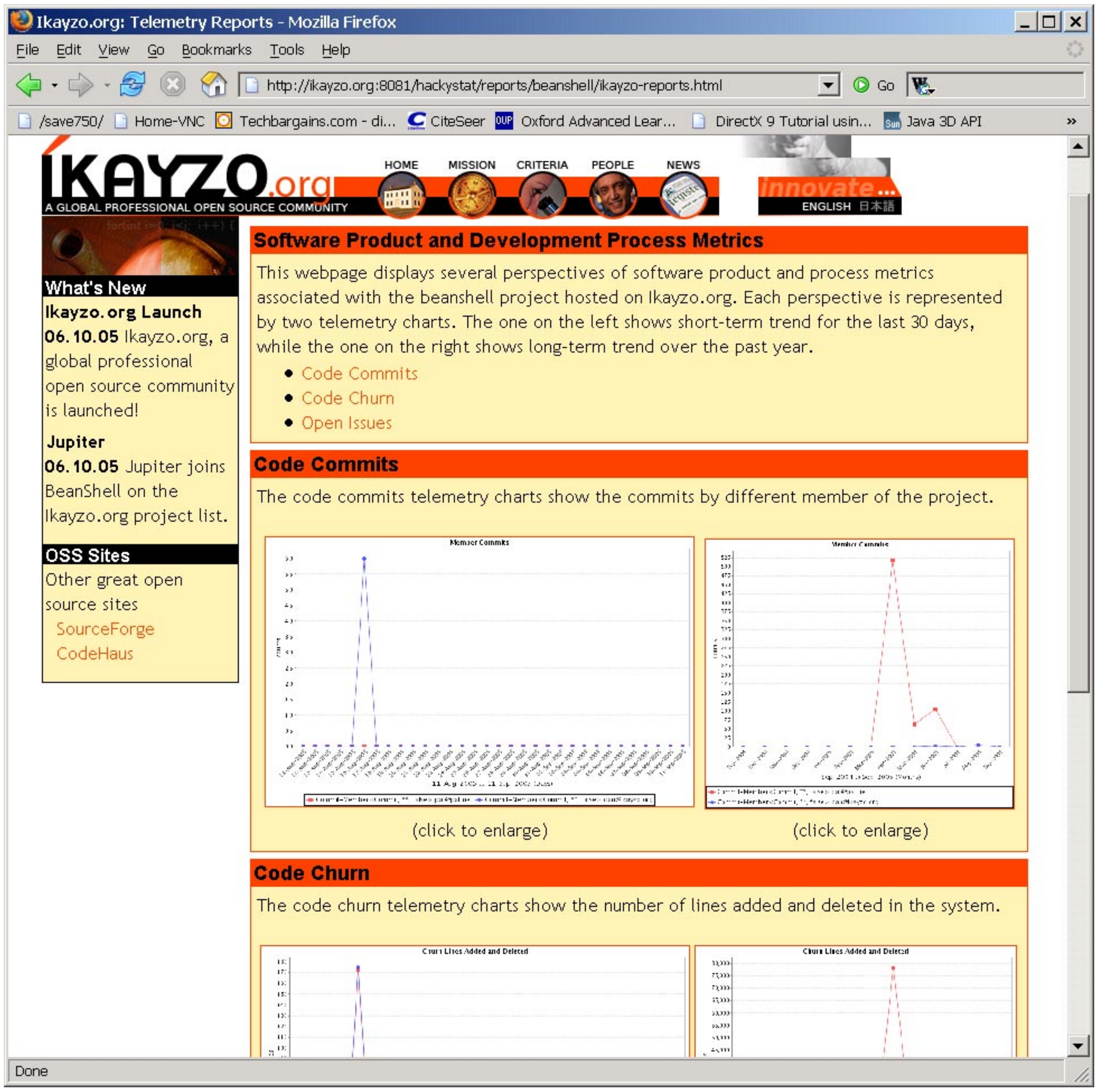
[1] http://www.ikayzo.org

and invoke telemetry analysis themselves, the software project telemetry system is hidden at Ikayzo in order to completely eliminate metrics collection and analysis overhead from the developers. I installed the sensors and pre-configured several telemetry charts. A background daemon process runs the telemetry analyses and updates the charts automatically every night. Users access telemetry metrics analysis results through a link on the web portal for each project. Figure 8.1 shows the exposed telemetry analysis page for one of the projects.

Since the preliminary investigation by Ikayzo and me indicates that most developers are cautious about having their personal software process data being collected, such as which files they are editing at what time, we decided to only install server-side sensors. "Server-side sensors" refers to those sensors that monitor activities of various server systems, such as version control server, bug tracking and issue management server. Currently the following metrics are collected: (1) source code commit metrics, and (2) bug and issue metrics. We are planning to add source code size metrics in the near future. These metrics are all public knowledge because everybody can check out source code and browse project issue database through the web. What the software project telemetry system does is essentially make these aspects of the development process more transparent using telemetry charts.

The other services provide by Ikayzo are very much like those provided by other project hosting sites such as *sourceforge.net* or *java.net*, except that Ikayzo is much smaller in scale. As of this writing three projects are hosted by Ikayzo:

- **BeanShell** is a small embeddable interpreter for a Java syntax compatible scripting language for the Java platform. The project has passed JSR-274 voting process, and is heading toward getting included in the Java Standard Edition at some point in the future.
- **Jupiter** is an Eclipse plug-in that provides code review and reporting functionality. It allows management of code annotations stored in XML based files, which can be shared by a development team.
- **SDL** stands for simple declarative language. It aims to provide an easy way to describe lists, maps, and trees of typed data in a compact, easy to read representation. Due to its simple API, it is a compelling alternative to XML for property files, configuration files, logs and simple serialization requires.



Ikayzo.org: Telemetry Reports - Mozilla Firefox

http://ikayzo.org:8081/hackystat/reports/beanshell/ikayzo-reports.html

What's New

Ikayzo.org Launch

06.10.05 Ikayzo.org, a global professional open source community is launched!

Jupiter

06.10.05 Jupiter joins BeanShell on the Ikayzo.org project list.

OSS Sites

Other great open source sites

SourceForge

CodeHaus

Software Product and Development Process Metrics

This webpage displays several perspectives of software product and process metrics associated with the beanshell project hosted on Ikayzo.org. Each perspective is represented by two telemetry charts. The one on the left shows short-term trend for the last 30 days, while the one on the right shows long-term trend over the past year.

- Code Commits
- Code Churn
- Open Issues

Code Commits

The code commits telemetry charts show the commits by different member of the project.

(click to enlarge) (click to enlarge)

Code Churn

The code churn telemetry charts show the number of lines added and deleted in the system.

Done

Figure 8.1. Telemetry Reports for Ikayzo Hosted Project

8.2 Case Study Design and Strategies

Open-source software development environments differ from more traditional centralized software development environment in that development is performed through volunteer work, which makes it impossible to mandate what kind of technology should be used in development process. Both my previous experience and initial investigation at Ikayzo suggest that developers are cautious about their personal process data being collected. As a result, the approach employed at Ikayzo is to avoid those sensitive personal process data. The software project telemetry system is used only to collect and analyze metrics from publicly available source such as project source code repository and issue tracking system, and make these aspects of the development process more transparent by presenting them in a more useful telemetry format.

The goal of this case study is multiple:

- To understand the adoption barrier of software project telemetry in an open-source environment where software development is collaborated through geologically-dispersed volunteer work and the decision-making process is decentralized.
- To observe how software project telemetry is used in open-source development and find out what works and what does not work.
- To understand both technical and social constraints of collection and analysis of different types of metrics in such an environment.

The evaluation in this case study will adopt the mixed methods paradigm. Since this will be largely a naturalistic study exploring how people use software project telemetry in an open-source development environment, priority will be given to qualitative information obtained during the course of this case study.

The two most popular approaches in qualitative data collection are survey and ethnographic observation. Though ethnography has the advantage of enabling to observe how developers actually interact with software project telemetry and make project management and process improvement decisions, it is not chosen because of the fact that the open-source developers under this case study contribute to the projects through volunteer effort at their personal time from different locations. As a result, survey method is the only feasible choice to gather feedback.

The decision to use survey method implies that I have to rely on the developers' self-reported opinions to evaluate software project telemetry. This threat can be mitigated by reconciling survey results with other data. Another source of data is quantitative information from telemetry report request log. If I find evidence that the developers look at telemetry report regularly, then I can put more confidence in their answers when they say software project telemetry is useful or not useful.

8.3 Data Collection and Analysis Procedures

This case study will collect data from two sources:

- Qualitative data from surveys of the developer's opinion about software project telemetry.
- Quantitative information about telemetry report requests.

The two sources of data will be integrated at data interpretation phase with emphasis on qualitative information and quantitative data corroborating the qualitative findings.

8.3.1 Surveys

Different approaches can be used to conduct a survey such as questionnaire or interview. In this case study, email questionnaires will be used for BeanShell project, and interviews will be used for Jupiter and SDL project. The reason is that the BeanShell project members are on the mainland and I don't have a chance to meet them in person, while I am able to meet other project members on a regular basis.

The survey, regardless of the approach chosen, will be conducted on a monthly basis. The purpose is to find out the developer's opinion about software project telemetry. The survey will follow the following basic steps:

- Ask the developers how often they view telemetry reports.
- Review telemetry reports together with the developers (omitted in email questionnaire).
- Find out what they think when they see the telemetry reports, and ask them whether the reports reflects the overall direction of the projects or not.

- Ask them what other telemetry analysis might be useful to them.
- Gradually introduce other available sensors and find out whether people are interested and under what condition they will use those sensors.
- Gradually reveal the regular telemetry analysis interface hidden underneath Ikayzo project portal which allow people to experiment with metrics themselves, and find out whether they exhibit interest beyond the predefined telemetry reports.

8.3.2 Telemetry Report Requests

Telemetry reports are integrated into the web portal for each project at Ikayzo. There are links on the main web page leading to telemetry charts. I will turn on web server logging facilities to log all telemetry chart requests. The information logged will include requester IP address, request time, and the telemetry chart being requested. The log will tell me the frequency of different telemetry charts being requested.

8.4 Researcher's Role

Ikayzo uses software project telemetry system to provide automated metrics collection and analysis service to the projects hosted on its site. The system is designed and developed by me as part of my dissertation research. It is distributed under GPL license and everyone can use it free of charge.

I provide volunteer technical support for Ikayzo running and administering its installation of software project telemetry system. My responsibility includes:

- Setting up server-side telemetry sensors for automated metrics collection.
- Configuring software project telemetry system.
- Defining telemetry reports to present metrics analysis results.
- Writing scripts to update and publish telemetry reports automatically.
- Monitoring the sensors and the system to make sure everything is working correctly.

My involvement in Ikayzo is strictly confined to providing software project telemetry system technical support. I don't provide support for other Ikayzo services, such as version control, bug tracking, and wiki discussion board. I don't participate in Ikayzo internal administrator. Finally, neither Ikayzo nor I have influence on the decision-making process of the hosted projects.

8.5 Threats, Verification, and Validation

8.5.1 Measurement Validity

Possible threat to measurement validity might come from qualitative data with respect to the developer's opinion of software project telemetry, but the risk is low. The reasons are as follows:

- Neither Ikayzo nor I can influence the decision-making process and the overall direction of the hosted projects. The developers make volunteer contribution to project development. They don't get paid. There is no incentive for them to misrepresent their true opinions toward software project telemetry.
- Three open-source projects are involved in this case study. The developers' opinion from different projects will be compared and contrasted.
- All telemetry report requests are logged. The quantitative information will be used to corroborate the qualitative findings.

8.5.2 Internal Validity

Internal validity refers to causality. In this case study, it is related to whether the telemetry reports have made the developers more aware of the status of the projects and thus contribute to better decision-making. Though this case study relies on the developers' subject opinion about the utility of software project telemetry, the threat is mitigated through the use of multiple data sources: data from 3 different projects, and quantitative and qualitative information from the the same project. All information will be reconciled and cross-validated at data analysis stage.

8.5.3 External Validity

External validity concerns the extent to which the experience gathered from the 3 projects hosted at Ikayzo can be generalized to other open-source development environment. The threat is quite low in this case study since the services provided by Ikayzo are pretty standard compared to other open-source project hosts such as *sourceforge.net* or *java.net*.

8.6 Expected Results

Not all metrics can be collected in an open-source development environment. Some might be feasible to collect technically, but infeasible socially. The case study will allow me to better understand the constraints associated with metrics collection and analysis. The experience gathered in this case study will enhance my understanding of software project telemetry technology adoption barriers, not only in open-source development environments, but also in more closed traditional software development environments.

Chapter 9

Conclusion

This research introduced *software project telemetry*, which is a novel, light-weight measurement approach. It includes both (1) highly automated measurement machinery for metrics collection and analysis, and (2) a methodology for in-process, empirically-guided software development process problem detection and diagnosis. In this approach, sensors collect software metrics automatically and unobtrusively. Metrics are abstracted in real time to telemetry streams, charts, and reports, which represent high-level perspectives on software development. Telemetry trends are the basis for decision-making in project management and process improvement. It overcomes many of the difficulties in existing approaches.

9.1 Anticipated Contributions

The anticipated contributions of this research are:

- The concept of software project telemetry as an effective automated approach to in-process, empirically-guided software development process problem detection and diagnosis.
- An implementation of software project telemetry which allows continuous monitoring of software project status, as well as generating and validating software process improvement hypothesis.
- The insights gained from the case studies regarding how to use software project telemetry effectively for project management and process improvement, as well as the adoption barrier of the technology.

9.2 Future Directions

There are several areas I am unable to address within the time frame of this dissertation research. They will be future directions:

- *Telemetry analysis user interface improvement* — Preliminary evaluation results suggest that the usability improvement to the interface invoking telemetry analysis is desired, especially with respect to the way telemetry analysis parameter values are supplied.
- *Telemetry language measurement scale type checking* — Current telemetry language does not enforce measurement scale type checking, and meaningless mathematical operations can be applied to telemetry streams as a result. The actual usage of the telemetry language needs to be studied to determine whether it is a big problem to end users or not. If it is, then the language needs to be augmented to account for scale type difference.
- *Statistic process control* — Given a set of telemetry streams, human judgment is required to detect bad trends in the development process. Statistic process control might provide automated support and needs to be studied in the context of software project telemetry.
- *Replication of case study* — The case studies need to be replicated in different settings in order to address external validity issues.
- *User base expansion* — One of the advantage of software project telemetry is that system deployment requires very little resource. This means the technology adoption risk is quite low for a software organization. I will make user interface improvements and try to find opportunities to market the technology.

Appendix A

Software Project Telemetry Language Specification (Version 1.2)

This document describes the syntax, semantics, and design of Software Project Telemetry Language. The language is neutral, and independent of any particular implementation.

A.1 Introduction

Software Project Telemetry Language is a language that allows user to specify rules to:

- define telemetry streams from software metrics,
- define telemetry chart from telemetry streams,
- and define telemetry report from telemetry charts.

Correspondingly, three primitive types are supported by the language: *streams*, *chart*, and *report*. The building blocks of telemetry *streams* are telemetry reducers, which are responsible for synthesizing and aggregating software metrics. Reducer returns a collection of telemetry streams, which is denoted as *streams* object in this specification. It is possible that the collection contains only one single stream. *streams* object can participate in mathematical operations. For example, you can add two *streams*, and the result is a new *streams* object. Telemetry *chart* defines the grouping of multiple telemetry *streams* into one single chart, while telemetry *report* specifies the grouping of multiple charts. The language supports parameterization.

This specification does not prescribe any reducers. The set of available reducers depends on particular implementation of the language. A reducer invocation looks like:

```
    ReducerName(parameter1, parameter2, ..., parameterN)
```

The language interpreter simply packs the parameters in an array, and passes them to the invocation of the reducer. It is reducer's responsibly to determine whether the parameters are valid or not, as well as the meaning of the parameters. The relationship between the language and telemetry reducers is like that of C language and its library functions. The difference is that you can write new functions in C but you cannot write new reducers using software project telemetry language. In other word, the reducers have to be supplied by the language implementation.

A.2 Getting Started

This section uses several examples to illustrate the essential features of the language, while later sections describe rules and exceptions in a detail-oriented and sometimes mathematical manner. This section strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with an introduction to the language. Note that the examples in this section uses several reducers, which may or may not be available in the particular implementation you are using.

A.2.1 Telemetry *Streams* Definition

The following statement defines a telemetry *steams*:

```
streams ActiveTime() = {
  "Active Time", "Hours", ActiveTime("**", "true")
};
```

- "streams" is the key word of the language.
- "ActiveTime" is the name of the telemetry *streams* defined by this statement.
- The contents in the curly braces is the body of the definition.

- The first part is the description.
- The second part is the invocation of a reducer called “ActiveTime”. Don’t confuse this with the name of the telemetry *streams* being defined.

A more complicated telemetry *streams* definition is presented below:

```
streams CodeChurn() = {
  "Lines addes plus deleted", "Lines",
  CodeChurn("LinesAdded") + CodeChurn("LinesDeleted")
};
```

“CodeCode” reducer returns either lines added or lines deleted. The two telemetry streams are added together to get a new telemetry stream about total code churn.

A.2.2 Telemetry *Chart* Definition

Suppose that we have ”streams” *A*, *B*, and *C* defined, then

```
chart mychart() = {
  "Chart Title", A, B, C
};
```

defines a chart containing *A*, *B*, and *C*. Note that there is no need to specify the label for x axis of the chart, since it is always the intervals represented by the telemetry streams.

A.2.3 Telemetry *Report* Definition

Suppose that we have “chart” *X*, *Y*, and *Z* defined, then

```
report myreport() = {
  "Report Name", X, Y, Z
};
```

defines a report containing charts *X*, *Y*, and *Z*.

A.2.4 Parameterization Support

The language supports position based parameters. An example is provided below:

```
streams MyStreams(member, filePattern1) = {
  "project member active time", "Hours",
  MemberActiveTime(filePattern1, "true", member)
};

chart  MyChart(filePattern2) = {
  "my own active time chart",
  MyStreams("me", filePattern2)
};

report MyReport(filePattern3) = {
  "my own report",
  MyChart(filePattern3)
};

draw R("**");
```

member, *filePattern1*, *filePattern2*, *filePattern3* are parameters. The definition of "MyChart" is interesting. It only instantiates one of the parameters. The final reducer invoked is:

```
    MemberCodeChurn("**", "true", "me")
```

A.3 Grammar

This chapter defines the lexical and syntactic structure of Software Project Telemetry Language. The grammars are presented using productions. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented lines contain a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. When there is more than one possible expansion, the alternatives are listed on separate lines preceded by "|". When there are many alternatives, the phrase *one of* may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line.

A.3.1 Lexical Grammar

The lexical grammar is presented in this section. The terminal symbols of the lexical grammar are the characters in the Unicode character set, and the lexical grammar specifies how characters are combined into tokens and white space. The basic elements that make up the lexical structure are *line terminators*, *white space*, and *tokens*. Of these basic elements, only tokens are significant in the syntactic grammar. Comments are not supported in this version of the language specification. The lexical processing consists of reducing telemetry language instances into a sequence of tokens which become the input to the syntactic analysis. Line terminators and white space have no impact on the syntactic structure, they only serve to separate tokens. When several lexical grammar productions match a sequence of characters, the lexical processing always forms the longest possible lexical element.

Line Terminators

Line terminators divide the characters into lines.

```
new-line:
    Carriage return character (U+000D)
  | Line feed character (U+000A)
  | Carriage return character followed by line feed character
  | Line separator character (U+2028)
  | Paragraph separator character (U+2029)
```

White Space

White space is defined as any character with Unicode class Zs which includes the space character, plus the horizontal tab character, the vertical tab character, and the form feed character.

```
whitespace:
    Any character with Unicode class Zs
  | Horizontal tab character (U+0009)
  | Vertical tab character (U+000B)
  | Form feed character (U+000C)
```

Tokens

There are several kinds of tokens: keywords, operators, punctuators, identifiers, and literals.

```
keywords: one of
  streams chart report draw

operator: one of
  = + - * /

punctuator: one of
  , ; ( ) { } "

identifier:
  [letter][letter|digit|-|_]*

string-literal:
  anything enclosed in double quotes

constant-literal:
  [1-9][digit]*

letter:
  [a-zA-Z]

digit:
  [0-9]
```

A.3.2 Syntactic Grammar

The syntactic grammar is presented in this section. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined. Two special tokens are used: $<$*EOF*$>$ denotes the end of input, and $<$*NULL*$>$ means nothing is required.

```
input:
    statements <EOF>

statements:
    statement
  | statements statement
```

```
statement:
    streams-statement ;
  | chart-statement ;
  | report-statement ;
  | draw-command ;


streams-statement:
    streams identifier ( function-declaration-parameter-list )
    = { streams-description , streams-unit-label , streams-definition }

streams-description:
    string-literal

streams-unit-label:
    string-literal

streams-definition:
    expression


chart-statement:
    chart identifier ( function-declaration-parameter-list )
    = { chart-title , chart-definition }

chart-title:
    string-literal

chart-definition:
    streams-reference
  | chart-definition , streams-reference

streams-reference:
    identifier ( function-reference-parameter-list  )


report-statement:
    report identifier ( function-declaration-parameter-list )
    = { report-title , report-definition }

report-title:
    string-literal

report-definition:
    chart-reference
  | report-definition , chart-reference

chart-reference:
    identifier ( function-reference-parameter-list  )


draw-command:
    draw identifier ( function-invocation-parameter-list )


expression:
    additive-expression

additive-expression:
    multiplicative-expression
  | additive-expression + multiplicative-expression
  | additive-expression - multiplicative-expression

multiplicative-expression:
    unary-expression
  | multiplicative-expression * unary-expression
  | multiplicative-expression / unary-expression

unary-expression:
    primary-expression
  | + unary-expression
  | - unary-expression

primary-expression:
    constant-literal
  | reduction-function
  | ( expression )

reduction-function:
    identifier ( function-reference-parameter-list )


function-declaration-parameter-list:
    <NULL>
  | template-parameter
  | function-declaration-parameter-list , template-parameter

function-invocation-parameter-list:
    <NULL>
  | resolved-parameter
  | function-invocation-parameter-list , resolved-parameter

function-reference-parameter-list:
    <NULL>
  | template-parameter
  | resolved-parameter
  | function-reference-parameter-list , template-parameter
  | function-reference-parameter-list , resolved-parameter

template-parameter:
    identifier

resolved-parameter:
    constant-literal
  | string-literal
```

A.4 Arithmetic Operations

Arithmetic operations involving telemetry ”streams” objects are valid in the following situations:

- Between two *streams* objects

  Arithmetic operations are valid so long as two ”streams” objects have the same number of telemetry streams, and the data points in those telemetry streams are all derived from the same intervals. Arithmetic operations are carried out between the individual data points in the corresponding interval. Note that it is up to the language implementation to determine how to match the telemetry streams in one ”streams” object to the telemetry streams in the other ”streams” object. For example, a particular implementation can attach tag to individual telemetry stream, and only allow streams with the same tag to be matched. If the implementation determines that a mapping cannot be found, the implementation is free to raise exceptions.

- Between one *streams* object and one constant

  The arithmetic operations in this situation are always valid. Each data point in the *streams* object participate in the operation with the constant individually, and the result is a new *streams* object.

A.5 Reducer

Reducer returns a collection of telemetry streams, and each stream represents one perspective on development process for a specific project during some specific time periods. Therefore, reducer invocation request is never complete without project and time interval information. However, this language specification does not prescribe how those information should be passed to reducers. The language implementation can pass the information as reducer parameters, or it can use some out-of-band mechanism to pass such information, such as storing the information in the context of user interaction.

A.6 Version History

Version 1.0 — The initial release of the software project telemetry language specification. June 2004.

Version 1.1 — Add parameterization support. March 2005.

Version 1.2 — Add multi-axis chart support. June 2005.

Appendix B

Software Project Telemetry End User Guide

This is the user guide for the reference implementation of software project telemetry based on the Hackystat framework. The actual user guide is omitted from this thesis proposal, but an online version is available at:

http://hackystat.ics.hawaii.edu/hackystat/docbook/ch05.html

Appendix C

Software Project Telemetry Reducer Developer Guide

The reference implementation of software project telemetry is based on the Hackystat framework. It offers a dynamic loading mechanism of additional telemetry reducers. This documentation is intended for developers who want to implement custom reducers. It assumes that you have a basic understanding of Hackystat source code. The basic steps of implementing a telemetry reducer are:

- Implement *org.hackystat.app.telemetry.processor.reducer.TelemetryReducer* interface.
- Write a configuration file, and name it *telemetry.<your-custom-name>.xml*.
- Copy the configuration file to *WEB-INF/telemetry* directory during build time.

C.1 Telemetry Reducer Interface

All reducers must implement *TelemetryReducer* interface. There is only one method in this interface:

```
TelemetryStreamCollection compute(Project project,
                Interval interval, String[] options)
    throws ReductionException;
```

which generates telemetry streams from the project for the specified interval. "Options" are reducer-specific parameters, which provides additional information to the reduction function. If user does not specify any parameter in telemetry definition, either null or an empty string array may be passed to the function. The return value is an instance of *TelemetryStreamCollection*.

C.2 Telemetry Reducer Configuration file

The xml configuration file must follow the template below:

```
<TelemetryReducers>
  <Reducer name="Reducer Name"
        class="Fully Qualified Implementing Class"
        reducerDescription="Description of this reducer"
        optionDescription="Description of optional parameters"
  />
  <!-- more "Reducer" elements can be put here. -->
</TelemetryReducers>
```

If there are more than one reducer, multiple *Reducer* elements can be put into the configuration file. More formally, it must conform to the following schema:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema targetNamespace="http://hackystat.ics.hawaii.edu
                            /telemetry/reducer.xsd"
           elementFormDefault="qualified"
           xmlns="http://hackystat.ics.hawaii.edu
                  /telemetry/reducer.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="TelemetryReducers">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Reducer" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="name"
                          type="xs:string" />
            <xs:attribute name="class"
                          type="xs:string" />
            <xs:attribute name="reducerDescription"
                          type="xs:string" />
            <xs:attribute name="optionDescription"
                          type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The configuration file name must follow the pattern *telemetry.**.xml*, and must be globally unique. It must be deployed to *WEB-INF/telemetry* directory during the build process. When the system starts up, *TelemetryReducerManager* scans directory *WEB-INF/telemetry* for files whose name matches the pattern *telemetry.**.xml*, and instantiates reducer instances defined in the configuration files. There will be one and only one instance for each reducer, therefore it is imperative that the *TelemetryReducer* implementation be thread-safe.

C.3 Performance

Hackystat telemetry infrastructure does not cache telemetry streams generate by telemetry reducers. If performance is critical, each reducer implementation should implement its own cache strategies.

Appendix D

Software Project Telemetry Survey in Classroom Setting

This is the survey distributed to the senior-level undergraduate and introductory-level graduate software engineering students at University of Hawaii. The classroom setting, the context, and the result of this survey is discussed in Chapter 6.

Hackystat Software Project Telemetry Survey

In order to better understand the strengths and weaknesses of **software project telemetry** and its current implementation based on the Hackystat framework, please take a few minutes to complete this survey. There is no right or wrong answer: I just want to know your opinions. Your opinions are important to this research.

Privacy statement: This survey is anonymous. I will not start processing survey results until after your grades have been reported. All individual data will be kept confidential, and only aggregated data will be published.

Part I: Background

Please give your best answer to the following questions. Your answers do not have to be exact.

1.	Which course are you taking now, ICS 413, 414, 613?	_____________
2.	How many years of programming experience do you have? (All programming experience counts, **including** the first ?Hello World? application.)	________ years
3.	How many years of paid professional programming experience do you have? (Please **exclude** half-time or less than half-time on-campus jobs while you are a student, such as student helper, and research assistant, even if you are paid to program.)	________ years

☞ *Please go to the next page.*

Part II: Multiple Choices

Please circle the number that most closely matches your feelings about the following statements. Choose ?N/A? if the statement does not apply. If you want to provide additional information, please use the comment line.

		Strongly Disagree	?	Neutral	?	Strongly Agree	
1.	I have no trouble installing and configuring Hackystat sensors. *Comment:* ______	1	2	3	4	5	N/A
2.	After Hackystat sensors are installed and configured (i.e. ignoring installation and configuration effort), there is no overhead collecting metrics. *Comment:* ______	1	2	3	4	5	N/A
3.	It's simple to invoke pre-defined Hackystat telemetry chart/report analyses. *Comment:* ______	1	2	3	4	5	N/A
4.	Hackystat telemetry analyses have shown me valuable insight into my and / or my team?s software development process. *Comment:* ______	1	2	3	4	5	N/A
5.	Telemetry analyses have helped me improve my software development process. *Comment:* ______	1	2	3	4	5	N/A
6.	If I was a professional software developer, I will want to use telemetry analyses in my development projects. *Comment:* ______	1	2	3	4	5	N/A
7.	If I was a project manager, I will want to use telemetry analyses in my development projects. *Comment:* ______	1	2	3	4	5	N/A

☞ *Please go to the next page.*

Part III: Free Response

Please use the following space to supply additional comments. It can be anything, such as concerns when using the system, suggestions on how to improve software project telemetry, etc. Thank you.

Thank you.

Bibliography

[1] C. Abts, B. W. Boehm, and E. B. Clark. COCOTS: A COTS software integration lifecycle cost model - model overview and preliminary data collection findings. In *ESCOM-SCOPE Conference*, 2000.

[2] A. J. Albrecht and J. Gaffney. Software function, source lines of code and development effort prediction. *IEEE Transactions on Software Engineering*, 1983.

[3] Ant. http://www.apache.org.

[4] E. Babbie. *Survey research methods*. Wadsworth, 1990.

[5] R. D. Banker, R. J. Kauffman, and R. Kumar. An empirical test of object-based output measurement metrics in a computer aided software engineering (case) environment. *Journal of Management Information Systems*, 1991.

[6] R. D. Banker, R. J. Kauffman, C. Wright, and D. Zweig. Automating output size and reuse metrics in a repository-based computer-aided software engineering (case) environment. *IEEE Transactions on Software Engineering*, 1994.

[7] V. Basili, L. Briand, S. Condon, Y. M. Kim, W. L. Melo, and J. D. Valett. Understanding and predicting the process of software maintenance releases. *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[8] V. R. Basili. Software modeling and measurement: The goal question metric paradigm. Technical Report CS-TR-2956, University of Maryland, College Park, 1992.

[9] V. R. Basili and H. D. Rombach. The TAME project: Towards improvement-oriented software environment. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

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

[11] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. D. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[12] J. Borstler, D. Carrington, G. Hislop, S. Lisack, K. Olson, and L. Williams. Teaching psp: Challenges and lessons learned. *IEEE Software, 19(5), September,* 2002.

[13] Bugzilla. http://www.bugzilla.org.

[14] C. H. Cherryholmes. Notes on pragmatism and scientific realism. *Educational Researcher*, 14:13–17, Aug-Sept 1992.

[15] S. Chulani and B. W. Boehm. Modeling software defect introduction and removal: COQUALMO. Technical Report usccse99-510, USC Center for Software Engineering, 1999.

[16] D. J. Clandinin and F. M. Connelly. *Narrative inquiry: Experience and story in qualitative research*. Jossey-Bass, 2000.

[17] ClearCase. http://www.rational.com.

[18] Clover. http://www.cenqua.com.

[19] Curise Control. http://cruisecontrol.sourceforge.net.

[20] Costar. Softstart Systems, http://www.softstarsystems.com.

[21] J. W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 2003.

[22] J. W. Creswell, V. P. Clark, M. Gutmann, and W. Hanson. *Handbook of mixed methods in the social and behavioral sciences*, chapter Advances in mixed method design. Saga Publications, 2003.

[23] CVS. http://www.cvsnt.org.

[24] M. K. Daskalantonakis. Achieving higher sei levels. *IEEE Software*, 1994.

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

[26] M. Diaz and J. Sligo. How software process improvement helped motorola. *IEEE Software*, 1997.

[27] R. Dion. Process improvement and the corporate balance sheet. *IEEE Software*, 10(4):28–35, July 1993.

[28] Eclipse. http://www.eclipse.org.

[29] K. E. Emam. The internal consistency of the ISO/IEC 15504 software process capability scale. In *5th. International Symposium on Software Metrics*, 1998.

[30] C. Fakharzadeh. CORADMO a software cost estimation model for RAD projects. In *16th International Forum on COCOMO and Software Cost Modeling*, 2001.

[31] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Publishing, 1997.

[32] P. Ferguson, W. S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya. Introducing the Personal Software Process: Three industry cases. *IEEE Computer*, 30(5):24–31, May 1997.

[33] A. Fuggetta, L. Lavazza, S. Marasca, S. Cinti, G. Oldano, and E. Orazi. Applying gqm in an industrial software factory. *ACM Transactions on Software Engineering and Methodology*, 1998.

[34] R. B. Grady and D. L. Caswell. Software metrics: Establishing a company-wide program. *Prentice-Hall*, 1987.

[35] W. Hayes and J. W. Over. The Personal Software Process: An empirical study of the impact of PSP on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Pittsburgh, PA, 1997.

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

[37] J. Herbsleb, D. Zubrow, D. Goldenson, W. Hayer, and M. Paulk. Software quality and the capability maturity model. *Communications of the ACM*, 1997.

[38] W. Hetzel. *Making Software Measurement Work: Building an Effective Software Measurement Program*. QED Publishing, 1993.

[39] W. S. Humphrey. Characterizing the software process. *IEEE Software*, 5(2):73–79, March 1988.

[40] W. S. Humphrey. *Managing the Software Engineering*. Addison-Wesley, 1989.

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

[42] W. S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[43] W. S. Humphrey, T. R. Snyder, and R. R. Willis. Software process improvement at hughes aircraft. *IEEE Software*, 1991.

[44] ISO. Iso 9001 quality systems - model for quality assurance in design/development, production, installation and servicing. *ISO*, 1987.

[45] ISO. Iso/iec tr 15504-1: 1998 information technology - software process assessment. *ISO*, 1998.

[46] JBlanket. http://csdl.ics.hawaii.edu.

[47] Jira. http://www.atlassian.com.

[48] JMeter. http://www.apache.org.

[49] P. M. Johnson. You cann't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications, Nashville, TN, December 2001.

[50] P. M. Johnson and A. M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6):85–88, November 1998.

[51] P. M. Johnson, H. B. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. 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.

[52] JUnit. http://www.junit.org.

[53] S. Khajenoori and I. Hirmanpour. An experiential report on the implications of the Personal Software Process for software quality improvement. In *The Fifth International Conference on Software Quality*, pages 303–312, 10 1995.

[54] P. Kulik and M. Haas. Software metrics best practices - 2003. *Technical report, Accelera Research Inc.*, 2003.

[55] P. Kuvaja, J. Simila, L. Krzanik, A. Bicego, G. Koch, and S. Saukonen. *Software Process Assessment and Improvement: the BOOTSTRAP approach*. Blackwell Publishers, 1994.

[56] F. Latum, R. Solingen, M. Oivo, B. Hoisl, D. Rombach, and G. Ruhe. Adopting gqm-based measurement in an industrial environment. *IEEE Software*, 1998.

[57] M. D. LeCompte and J.J. Schensul. *Designing and conducting ethnographic research*. AltaMira, 1999.

[58] LOCC. http://csdl.ics.hawaii.edu.

[59] Christoph Lofi. Continuous GQM. Master's thesis, Technical University Kaiserslautern, Germany, 2005.

[60] S. B. Merriam. *Qualitative research and case study applications in education*. Jossey-Bass Publishers, 1998.

[61] M. B. Miles and A. M. Huberman. *Qualitative data analysis: A sourcebook of new methods*. Saga Publications, 1994.

[62] C. A. Moore. Project LEAP: Personal process improvement for the differently disciplined. In *International Conference on Software Engineering*, pages 726–727, 5 1999.

[63] C. Moustakas. *Phenomenological research methods*. Sage Publications, 1994.

[64] J. P. Murphy. *Pragmatism: From Peirce to Davidson*. Westview Press, 1990.

[65] I. Newman and C. R. Benz. *Qualitative-quantitative research methodology: Exploring the interactive continum*. Southern Illinois University Press, 1998.

[66] M. Q. Patton. *Qualitative evaluation and research methods*. Saga Publications, 1990.

[67] M. C. Paulk, B. Curtis, M. B. Chrissis, and C.V. Weber. Capability maturity model, version 1.1. Technical Report CMU/SEI-93-TR-024, Carnegie Mellon Software Engineering Institute, 1993.

[68] W. C. Peterson. SEI's software process program - presentation to the board of visitors, 1997. Software Engineering Institute, Carnegie Mellon University.

[69] S. L. Pfleeger and C. McGowan. Software metrics in the process maturity framework. *The Journal of Systems and Software*, 1990.

[70] Estimate Professional. Software Productivity Center Incorporated, http://www.spc.ca.

[71] L. H. Putnam. QSM, Inc. http://www.qsm.com/.

[72] L. H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transaction on Software Engineering*, 4(4):345–361, July 1978.

[73] R. Rorty. *Pragmatism as anti-representationalism, in Pragmatism: From Peirce to Davidson, pp. 1-6*. Westview Press, 1990.

[74] L. Schatzman and A. L. Strauss. *Field research; strategies for a natural sociology*. Prentice-Hall, 1973.

[75] SEI. *The Capability Maturity Model: Guidelines for Improving Software Process*. Addison-Wesley, 1995.

[76] R. Solingen and E. Berghout. *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.

[77] R. Solingen, E. Berghout, and F. Latum. Interrupts: Just a minute never is. *IEEE Software*, 1998.

[78] Standish. The Chaos Report. http://www.standishgroup.com/sample_research/PDFpages/-chaos1994.pdf, 1994.

[79] A. Strauss and J. Corbin. *Basics of qualitative research: Grounded theory procedures and techniques*. Sage Publications, 1998.

[80] Visual Studio. http://www.microsoft.com.

[81] A. Tashakkori and C. Teddlie. *Mixed methodology: Combining qualitative and quantitative approaches*. Saga Publications, 1998.

[82] D. Tuma. Software process dashboard. http://processdash.sourceforge.net, 2000.

[83] Cost Xpert. Cost Xpert Group Incorporated, http://www.costxpert.com.