

IMPROVING
SOFTWARE DEVELOPMENT PROCESS AND PROJECT MANAGEMENT
WITH
SOFTWARE PROJECT TELEMETRY

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

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

DECEMBER 2006

By
Qin Zhang

Dissertation Committee:

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

We certify that we have read this dissertation and that, in our opinion, it is satisfactory in scope and quality as a dissertation for the degree of Doctor of Philosophy in Computer Science.

DISSERTATION COMMITTEE

Chairperson

©Copyright 2006

by

Qin Zhang

To my parents and my girl-friend.

Acknowledgments

I would like to thank Dr. Philip Johnson, my dissertation adviser, my mentor, who guided me through my research, who encouraged me in difficult times, and who even spent numerous hours proof-reading my manuscript. This research would not have been possible without you. You have greatly influenced my skill in software engineering and research methodology.

I would like to thank my fellow CSDL members: Hongbing Kou, Mike Paulding, Julie Sakuda, and Austen Ito. You helped me improve my research and make my work environment fun.

I would like to thank my committee members: Dr. Daniel Suthers, Dr. Martha Crosby, Dr. Wes Peterson, and Dr. Daniel Port. Your time and support have been priceless.

Lastly, I would like to thank the Department of Information and Computer Sciences, especially Dr. David Pager, who made my transition from Economics to Computer Science a reality.

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 software development processes and final products. However, due to the high cost associated with “metrics collection” and difficulties in “metrics decision-making,” measurement is not widely adopted by software organizations.

This dissertation proposes a novel metrics-based program called “software project telemetry” to address the problems. It uses software sensors to collect metrics automatically and unobtrusively. It employs a domain-specific language to represent telemetry trends in software product and process metrics. Project management and process improvement decisions are made by detecting changes in telemetry trends and comparing trends between different periods of the same project. Software project telemetry avoids many problems inherent in traditional metrics models, such as the need to accumulate a historical project database and ensure that the historical data remain comparable to current and future projects.

The claim of this dissertation is that software project telemetry provides an effective approach to (1) automated metrics collection and analysis, and (2) in-process, empirically-guided software development process problem detection and diagnosis. Two empirical studies were carried out to evaluate the claim: one in software engineering classes, and the other in the Collaborative Software Development Lab. The results suggested that software project telemetry had acceptably-low metrics collection and analysis overhead, and that it provided decision-making value at least in the exploratory context of the two studies.

Table of Contents

Acknowledgments	v
Abstract	vi
List of Tables	xii
List of Figures	xiii
1 Introduction	1
1.1 Software Crisis	1
1.2 Software Measurement, Process Improvement, and Predictions	2
1.3 Problem Statement	3
1.3.1 Metric Collection Cost Problem	3
1.3.2 Metrics Decision-Making Problem	4
1.4 Proposed Solution - Software Project Telemetry	5
1.4.1 Metrics Collection	6
1.4.2 Metrics Decision-making	6
1.4.3 Process Methodology	15
1.5 Thesis Statement	17
1.6 Empirical Evaluation	18
1.7 Contribution	20
1.8 Thesis Organization	22
2 Related Work	23
2.1 Software Measurement Theory	24
2.2 Personal Software Process	27

2.3	Constructive Cost Model and Model-based Process Predictions	29
2.4	Goal-Question-Metric Paradigm	32
2.5	Capability Maturity Model and Process Maturity Frameworks	33
2.6	Chapter Summary	37
3	Software Project Telemetry	38
3.1	Overview	39
3.2	Sensor-based Data Collection	40
3.3	Telemetry Language and Telemetry Constructs	42
3.4	Telemetry-guided Project Management and Process Improvement	45
3.5	Chapter Summary	47
4	Implementation	48
4.1	Hackystat Framework	50
4.1.1	Metrics Storage	50
4.1.2	Project Definition Management	51
4.1.3	Extension Mechanism	52
4.2	Hackystat Telemetry Module	52
4.2.1	Functional Description	52
4.2.2	Implementation Details	54
4.3	Telemetry Reducers and Functions	58
4.3.1	Telemetry Reducers	58
4.3.2	Telemetry Functions	60
4.4	Chapter Summary	61
5	Evaluation Strategy	62
5.1	Review of Research Methods	62
5.1.1	Quantitative Paradigm and Post-Positivism	62
5.1.2	Qualitative Paradigm and Constructivism	63
5.1.3	Mixed-Methods Paradigm and Pragmatism	66
5.1.4	Clarification of Terminologies	67

5.2	Software Project Telemetry Evaluation Design	68
5.3	Chapter Summary	72
6	Classroom Study	74
6.1	Classroom Setting	74
6.2	Researcher's Role	76
6.3	Study Design	76
6.4	Data Collection and Analysis	77
6.4.1	Survey Questionnaire	77
6.4.2	System Usage Log	78
6.5	Results	79
6.5.1	Results from Individual Survey Question	79
6.5.2	Results from Free Response Section	87
6.5.3	Results from Telemetry Analysis Invocation Log	90
6.6	Study Conclusion	93
7	CSDL Study	94
7.1	CSDL Setting	95
7.2	Researcher's Role	99
7.3	Study Design	100
7.4	Data Collection and Analysis	101
7.5	Results	105
7.5.1	Improvement on CSDL Release Cycle Issue Management	106
7.5.2	Improvement on CSDL Code Quality	112
7.5.3	Improvement on Developers' Insights into their Software Development Process	117
7.5.4	Top-down Telemetry Design	124
7.5.5	Sensor Verification	130
7.5.6	Limitation on Low-level Details	135
7.5.7	Telemetry Language Enhancement with Filter Functions	139

7.5.8	Telemetry Language Enhancement with Y-axis Construct	144
7.5.9	Runtime Performance Enhancement	149
7.6	Study Conclusion	153
7.6.1	Decision-Making Values	153
7.6.2	Insights	154
7.6.3	Telemetry System Improvement	155
8	Evaluation Conclusions	156
8.1	Synthesis of Study Results	156
8.1.1	Metrics Collection	157
8.1.2	Analysis Invocation	158
8.1.3	Decision Making	160
8.1.4	Best Practice	162
8.1.5	Adoption Barrier	162
8.2	Future Evaluations	163
9	Final Remarks	165
9.1	Research Summary	165
9.2	Dissertation Contribution	167
9.2.1	Concept of Software Project Telemetry	167
9.2.2	Implementation of Software Project Telemetry	168
9.2.3	Insights from Empirical Studies	169
9.3	Future Directions	170
A	Software Project Telemetry Language Specification	173
A.1	Introduction	173
A.2	Getting Started	174
A.2.1	Telemetry Report	175
A.2.2	Telemetry Chart and Y-axis	175
A.2.3	Telemetry Stream	176
A.2.4	Telemetry Reducer	177

A.2.5 Telemetry Function	177
A.3 Grammar	178
A.3.1 Lexical Grammar	178
A.3.2 Syntactic Grammar	180
A.4 Special Considerations	183
A.4.1 Arithmetic Operations	183
A.4.2 Telemetry Reducers and Functions	184
B Classroom Survey	185
C CSDL Data Summary	189
Bibliography	201

List of Tables

<u>Table</u>	<u>Page</u>
2.1 Software Measurement Classification	24
2.2 Measurement Scale and Valid Mathematical Operations	25
2.3 Process Maturity and Measurement	34
2.4 CMM Levels and Key Process Areas	36
7.1 Nash Equilibrium in a Non-Cooperative Game	121
8.1 Data Access Recommendations by Grady	163

List of Figures

<u>Figure</u>	<u>Page</u>
1.1 Release Issue Tracking: Total vs. Open Issues	7
1.2 Telemetry Report Analysis	9
1.3 Telemetry Expert Analysis	14
1.4 Telemetry Chart	16
2.1 Goal-Question-Metric Paradigm	33
3.1 Telemetry-based Process Improvement	46
4.1 Software Project Telemetry System Implementation	49
4.2 Telemetry Definition Management Console	53
4.3 Core_Telemetry Module Package Structure	56
4.4 Telemetry Language Abstract Syntax Tree	57
6.1 Telemetry Analysis Invocation by Month	91
6.2 Telemetry Analysis Invocation by Individual and Month	92
7.1 Hackystat Size by Programming Language	96
7.2 CSDL Software Development Process	97
7.3 Telemetry Control Center on Telemetry Wall	98
7.4 A Page with Links to all Raw Data Entries	103
7.5 One of the Raw Data Entries with Annotation	103

7.6	A Tables with Links to all Generated Hypotheses	104
7.7	One of the Generated Hypotheses	104
7.8	Hackystat Release Cycle 7.3 — Total Issues vs. Remaining Issues	110
7.9	Hackystat Release Cycle 7.3 — Total Issues vs. Active Time	110
7.10	Hackystat Release Cycle 7.4 — Total Issues vs. Remaining Issues	111
7.11	Hackystat Release Cycle 7.4 — Total Issues vs. Active Time	111
7.12	FindBugs and PMD Warnings from the Hackystat Source	116
7.13	FindBugs Warnings in “Fail” and “Monitor” Categories	116
7.14	Integration Build Failures and Process Metrics	123
7.15	Telemetry Report: Page 1	128
7.16	Telemetry Report: Page 2	129
7.17	Telemetry Chart Indicating “Emma” Sensor not Working	134
7.18	Telemetry Chart Indicating “Eclipse” Sensor not Working	134
7.19	CSDL Enhanced Version of FindBugs Report	138
7.20	Telemetry Chart with Unfiltered Module Coverage	142
7.21	Telemetry Chart with Filtered Module Coverage	143
7.22	Telemetry Charts with Automatically-scaled Vertical Axes	148
7.23	Telemetry Charts with Manually-specified Vertical Axes	148

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* [77] published by the Standish group, companies in the United States spent more than \$250 billion each year on IT application development of approximately 175,000 projects. Only 16% of these projects finished on schedule and within budget. Another 31% were cancelled before completion, mainly due to quality problems, for losses of about \$81 billion. Approximately 53% exceeded their original budgets by an average of 189% for losses of about \$59 billion. Those projects that managed to final 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 [67] from the Software Engineering Institute (SEI) indicated that out of 542 software organizations participating in the CMM maturity assessment, 67% of them were at CMM Level 1 (the lowest process maturity level), and 20% were at CMM Level 2. The software process at CMM Level 1 is characterized as *ad hoc* and sometimes *chaotic*. Inputs to the process are ill-defined, and

the transition from inputs to final software products is uncontrolled. The development process is so reactive that management control is impossible. The development process at CMM Level 2 is better, because earlier successes on projects with similar applications can potentially be repeated. However, there is still no visibility as to how software products are produced, and any disturbance to the development team or resources can easily cause project failure. In other words, 87% of the software organizations in the survey were unable to control their development processes. Nor were they able to consistently develop software products on schedule and within budget.

Uncontrollable and non-repeatable processes cause many problems for 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*” [22].” 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 aspects of processes and products more visible, and give us better understanding of the relationship 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 have controllable and repeatable development processes, and possess 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 [42, 43], CMM [41, 66, 42], ISO 9001 [45], SPICE [46, 27] and BOOTSTRAP [59]. Industrial experiences [36] have 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 pro-

cesses are essential for software organizations to make reliable predictions about their development activities, such as those in SLIM [72, 71] and COCOMO [10, 11].

1.3 Problem Statement

Despite the potential for software measurement in theory and positive experiences [36] in reality, effective application appears far from mainstream in practice. For example, a recent case study [58] surveyed 630 software professionals. It divided software development organizations into two groups: “best practice” and “all other.” 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 this number is just 2% for organizations in the “all other” category.

Research has identified a variety of reasons for this discrepancy. They can be categorized into two major groups:

- The Metrics Collection Cost Problem
- The Metrics Decision-making Problem

1.3.1 Metric Collection Cost Problem

All measurement activities compete for resources. An important question that a software organization committing itself to measurement program must answer is whether the benefit from measurement outweighs the cost. Existing measurement programs tend to be very expensive. For example, the PSP uses manual metrics collection. It is not only tedious, but also susceptible to bias, error, omission, and delay. The adoption of tool-based metrics collection techniques, such as LEAP [65], PSP Studio [38], and Software Process Dashboard [82], does not completely solve these problems because of their chronic overhead that requires the user to constantly switch back and forth between doing work and telling the tool what work is being done [51, 53]. CMM requires that measurement be taken in all key process areas in order to determine the status of the activities. Quantitative measurement is mandatory in CMM Level 4 and 5. Humphrey himself admitted that:

“The greatest potential problem with the managed process is the cost of gathering data, and that there are an enormous number of potentially valuable measures of the software process, but such data are expensive to gather and to maintain.” [40]

Due to the high cost associated with metrics collection, it is a daunting task to apply measurement best practices to improve a software organization’s development process in practice.

1.3.2 Metrics Decision-Making Problem

The metrics decision-making problem refers to the problem of how to make project management and process improvement decisions based on information in the metrics. Traditional approaches use a “historical project database” as a baseline for comparison with metrics from the current project. They typically involve the following 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 collected metrics, (3) 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, one 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 , and so forth.

This model-based process prediction technique is used in many forms, such as PSP (Section 2.2) and COCOMO (Section 2.3). The technique faces a number of limitations:

- First, the predictive power of the model 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. Apart from the cost of accumulating the historical project database, recalibration 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 the current project may differ from those in the historical project database.
- Second, model-based process prediction assumes that a software organization’s development process is predictable. However, according to the SEI survey [67], 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. It is simply impossible to make predictions for these organizations. As a result, the majority of

software development organizations are incapable of benefiting from model-based approaches to improve their development processes.

- Lastly, most models available are built to compare a set of finished projects. This may be useful for initial project planning. But if you want to manage a project that is still being developed, most models are not designed with such in-process control in mind. In other words, how do you use metrics from completed projects to manage a project 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, but few of them are capable of implementing a successful measurement program in practice. Many of the difficulties lie in one or both of the “*metrics collection cost*” and “*metrics decision-making*” problems.

1.4 Proposed Solution - Software Project Telemetry

In this thesis, I propose a novel approach to software measurement called “*software project telemetry*”. It addresses the “*metrics collection cost problem*” through highly automated measurement machinery: software sensors are written to collect metrics automatically and unobtrusively. It addresses the “*metrics decision-making problem*” through a domain-specific language designed for the representation of telemetry trends for different aspects of software development process.

In contrast to model-based approaches, which involve the comparison of data from different projects, software project telemetry focuses on the comparison of data taken from the same project at different times. This within-project data comparison involves a much smaller time scale: typically with intervals of days or weeks. The idea is that comparison can be made more effectively between two different periods of the same project than between two different projects. It thus avoids many problems a model-based approach suffers from, such as spending the cost of accumulating a historical project database first and then constantly worrying about whether the current project is comparable to those in the database. In software project telemetry, the metrics from the initial period of the project are used to establish a baseline and bootstrap the process. Project management and process improvement decisions are made by detecting changes in telemetry trends and comparing trends between two periods of the same project. In-process control for a project that is still

being developed is made possible precisely because comparisons are made within the same project instead of across projects.

1.4.1 Metrics Collection

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

Software process metrics are the metrics that assist in monitoring and controlling the way software is produced. 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 (1) a sensor for an IDE that monitors developer activities, such as code editing effort, compilation attempts and results, etc., or (2) a sensor for an integration build system that monitors the number of times the program failed to rebuild overnight.

Software product metrics are the metrics that describes the properties of the software itself. Sensors collecting product metrics are typically implemented as analyzers for software artifacts. An example is an analyzer that parses program source code to compute size and complexity information.

There are many possibilities for sensors and the data that they collect. However, the key point is that sensors are designed to collect metrics automatically and unobtrusively. This way, sensors not only keep metrics collection cost low, but also enable developers to focus on their primary task – developing software products instead of recording process and product metrics.

1.4.2 Metrics Decision-making

The metrics collected by sensors are time-stamped, and these time stamps are always significant in metrics analysis. Telemetry streams, charts, and reports capture high-level perspectives on software development, while the telemetry language facilitates the exploration of these perspectives in telemetry analyses. Telemetry analyses can be performed at different levels.

An example of relatively high-level telemetry analysis is release cycle issue tracking. Figure 1.1 displays two issue tracking charts for “Hackystat-7” project release cycle 7.3 and 7.4 respectively.

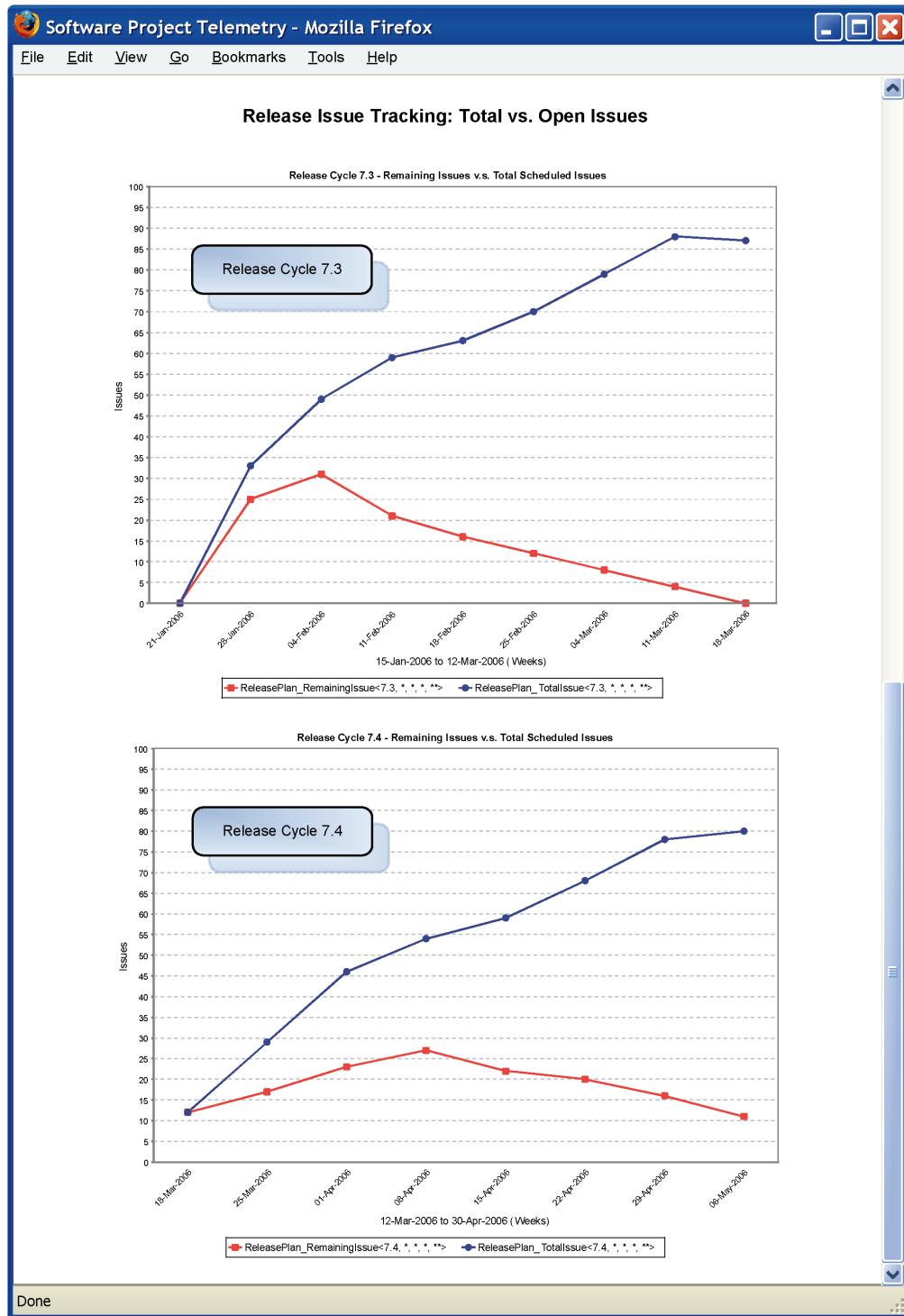


Figure 1.1. Release Issue Tracking: Total vs. Open Issues

Each chart shows the total and remaining number of issues on the last day of each week during the two release cycles. The top blue line represents a telemetry stream for the total number of issues scheduled for that release; while the red line below represents a telemetry stream for the number of remaining issues. The release cycle is complete when the number of remaining issues touches zero. An interesting process level observation from the perspective of project management is that the manager did not schedule everything up-front. Instead, he added new issues almost every week. Nevertheless, he was able to manage the team to make consistent progress toward zero open issue to finish the release cycle. The red line (i.e., the bottom line) provided information about the trend in issue closure that helped the manager assess whether or not more issues could be added to that release cycle. Another observation is that the telemetry charts in Figure 1.1 were generated when release cycle 7.4 was still in progress. By the end of May 6, 2006, there were still 11 open issues. From the perspective of project planning and scheduling, telemetry charts for previously finished release cycles, such as the chart on the top for release cycle 7.3, serve as the base line. By comparing the shape of telemetry streams in different release cycles, the project manager was able to make an in-process decision concerning whether the overall development process was stable, and estimate whether the team could finish the current release cycle on schedule.

Telemetry analysis can also be performed at a relatively low level to reveal details of software development process. An example of such an analysis is provided below, together with the introduction of basic telemetry language constructs: *stream*, *chart*, and *report*. Figure 1.2 illustrates their relationship.

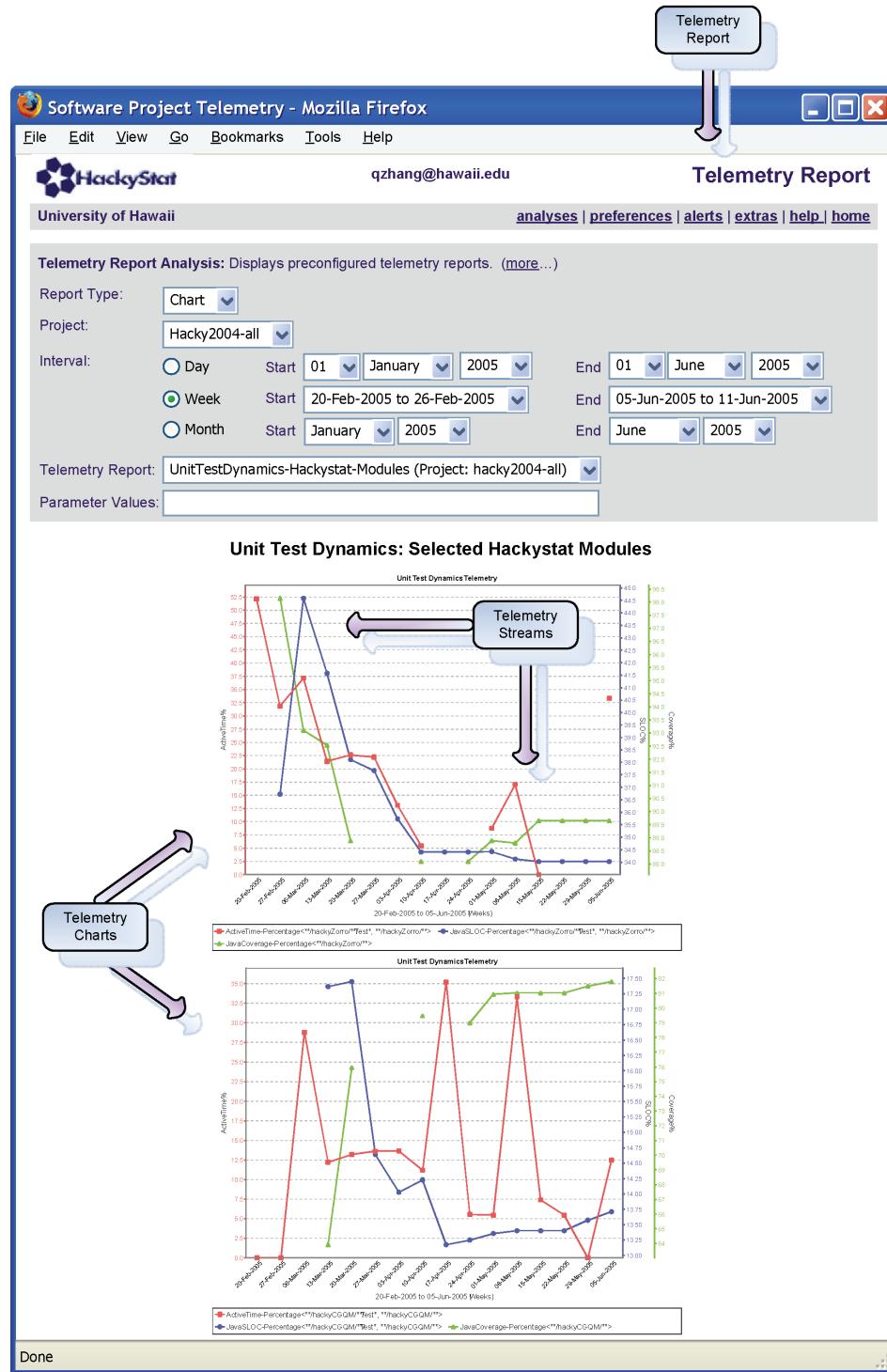


Figure 1.2. Telemetry Report Analysis

Telemetry 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.

For example, Figure 1.2 shows a telemetry report for “Hacky2004-all” project covering the time interval from the week of Feb 26, 2005 to the week of June 5, 2005. The report consists 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 (*ActiveTime-Percentage*) the percentage of source code devoted to testing (*JavaSLOC-Percentage*), and the percentage of test coverage that results from this effort and code (*JavaCoverage-Percentage*).

The two charts share the same time interval and project. The only difference is that they show unit test dynamics information for two different modules in the same project. Two developers are primarily responsible for the two modules respectively. Interestingly, the unit test dynamics telemetry trends for the two modules reveal a very different shape, indicating differences in the underlying approach to the development of the two modules. An important note is that software project telemetry does not presume any judgment as to which approach to development is better. Some projects might choose to trade a little testing for time-to-market. Others might require every single line of code perform exactly as intended.

Telemetry Chart

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, the same Figure 1.2 shows two instances of the same telemetry chart. Each chart contains three telemetry streams. 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

¹ Active time is a proxy for developer effort and is based upon measuring the time spent writing and editing code inside an IDE.

chart contains streams parameterized for the *hackyZorro* module, while the bottom chart contains streams parameterized for the *hackyCGQM* module.

Telemetry Stream

Telemetry streams are sequences of a single type of software process or product data for a single project over a specified time interval. They are best thought of as a kind of abstract data type representing one or more series of metric data values of the same type.

The time interval covered by a 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 telemetry data points:

- Aggregated information — The 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 each period.

For example, the time period used in the telemetry streams in Figure 1.2 is week. The data points in *ActiveTime-Percentage* telemetry stream represent aggregated information. They are the total time spent on editing source code for the entire week. The data points in *JavaSLOC-Percentage* and *Coverage-Percentage* telemetry streams represent snapshot information. They are the number of source code lines and system test coverage at the end of each week.

The advantage of telemetry is that it shows the history of some form of state in the project development environment, and helps the project manager detect changes in the development process. Another advantage is that it is generally tolerant of missing data. For example, there are missing dots in the telemetry streams in Figure 1.2. While complete data provide the best support for project management or process improvement, occasional drop-outs of data should have little impact on the value of telemetry 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.

Telemetry Language

Under the hood, telemetry reports, charts, and streams are generated using the telemetry language. The language serves two purposes:

- Different software development environments and different projects might have different requirements for metrics analysis. The telemetry language provides a flexible mechanism that decouples the types of metrics we collect and the types of analyses we support.
- Many interesting issues in software project management involve understanding the relationship between different metrics. For example, we might be interested in seeing whether an increased investment in code review pays off with less unit test failures, or increased test coverage, or less defects reported against the reviewed modules. The telemetry language enables interactive exploration of the relationship between metrics by allowing a user to experiment with the data to see what perspectives provides best insight into his/her particular situation.

The telemetry language that is used to generate the telemetry report in Figure 1.2 is listed below:

```
streams ActiveTime-Percentage(filePattern1, filePattern2) = {  
    "Active Time Percentage",  
  
    ActiveTime(filePattern1, "false")  
    / ActiveTime(filePattern2, "false")  
    * 100  
};  
  
streams JavaCoverage-Percentage(filePattern) = {  
    "Java Coverage Percentage",  
  
    JavaCoverage("Percentage", filePattern, "method")  
};  
  
streams JavaSLOC-Percentage(filePattern1, filePattern2) = {  
    "Java SLOC Percentage",  
  
    FileMetric("Java", "sourceLines", filePattern1)  
    / FileMetric("Java", "sourceLines", filePattern2)  
    * 100  
};
```

```

y-axis yAxis(label) = {label};

chart UnitTestDynamics-Chart(filePattern, testFilePattern) = {
    "Unit Test Dynamics Telemetry",
    (ActiveTime-Percentage(testFilePattern, filePattern),
     yAxis("ActiveTime%")),
    (JavaCoverage-Percentage(filePattern),
     yAxis("Coverage%")),
    (JavaSLOC-Percentage(testFilePattern, filePattern),
     yAxis("SLOC%"))
};

report UnitTestDynamics-Hackystat-Report() = {
    "Unit Test Dynamics: Selected Hackystat Modules",
    UnitTestDynamics-Chart("**/hackyZorro/**",
                           "**/hackyZorro/**/Test*"),
    UnitTestDynamics-Chart("**/hackyCGQM/**",
                           "**/hackyCGQM/**/Test*")
};

draw UnitTestDynamics-Hackystat-Report();

```

Two features of the language are illustrated in the example above:

- The telemetry language supports arithmetic operations. You can add, subtract, multiply, and divide two telemetry streams.
- The telemetry language supports parameterization. The two charts in the “UnitTestDynamics-Hackystat-Report” definition are the same except that they are passed two different parameter values representing two different modules in the project.

Figure 1.3 shows the telemetry analysis expert interface. The telemetry charts generated are exactly the same as those in Figure 1.2. The only difference is that this time we are using the telemetry language to interact with the system directly. Note, however, that the second chart is not shown in Figure 1.3 because of space constraint. The telemetry language specification in Appendix A contains more detailed information.

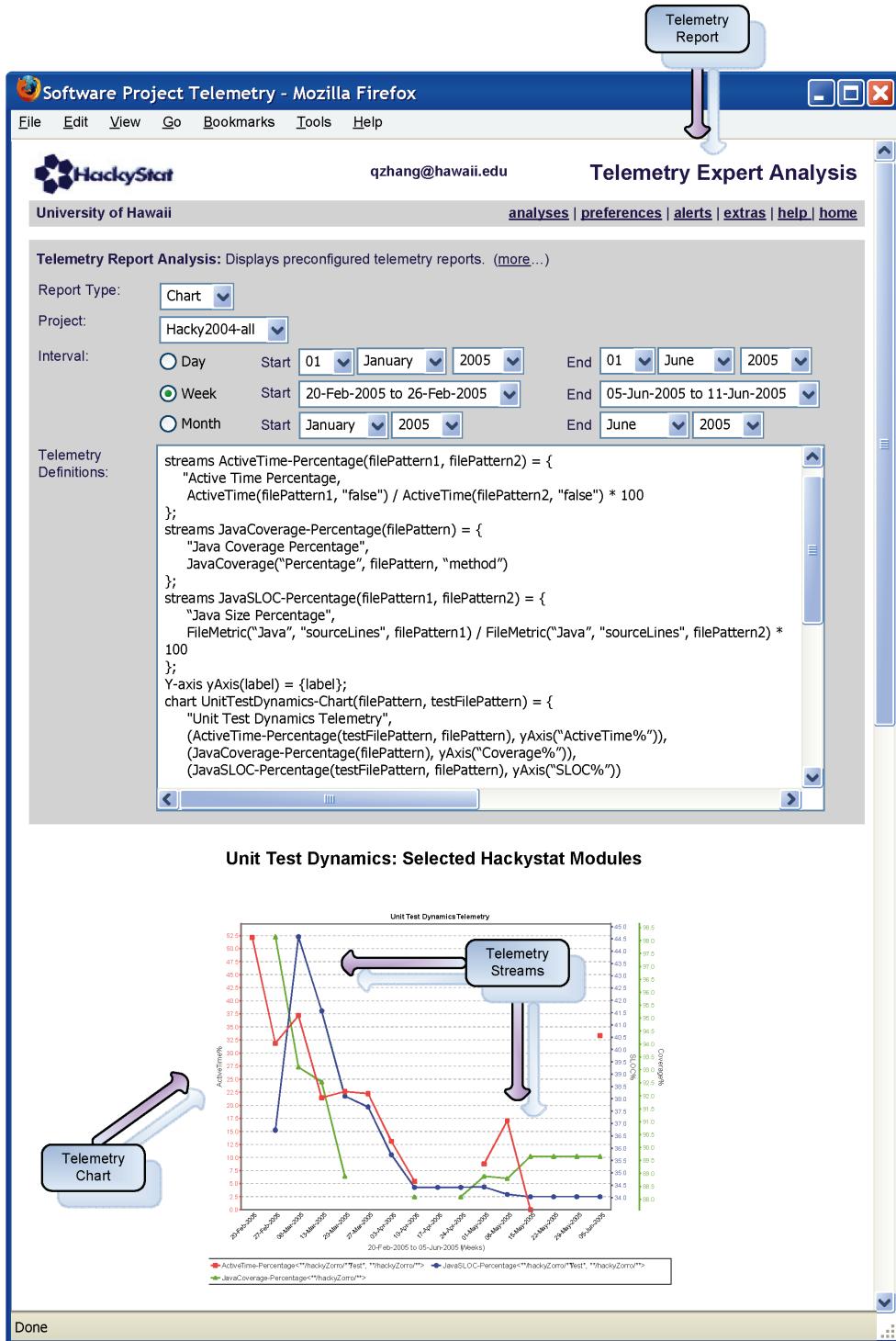


Figure 1.3. Telemetry Expert Analysis

1.4.3 Process Methodology

Figure 1.4 is an enlarged view of the top chart in Figure 1.2 and Figure 1.3, showing unit test dynamics for the *hackyZorro* module.

A *unit test* is a procedure used to verify that a particular module of source code is working correctly. This chart allows the comparison of the cost and the quality of unit testing. *ActiveTime-Percentage* is the percentage of code-editing time allocated to writing test cases, and *JavaSLOC-Percentage* is the percentage of source code lines that is unit test cases. Both of them are used as proxies for testing cost. *JavaCoverage-Percentage* is the resulting test coverage (i.e., the percentage of the system exercised by test cases). It is used as a proxy for testing quality. Ideally, we wish to see high testing quality but low testing cost. When displayed in telemetry chart, we want to see the *JavaCoverage-Percentage* telemetry stream near the top of the chart while the *ActiveTime-Percentage* and *JavaSLOC-Percentage* telemetry streams near the bottom of the chart.

The telemetry chart in Figure 1.4 shows the development of the *hackyZorro* module in the project. Both testing cost and testing quality were high at the beginning, but both were decreasing over time. After a little investigation, it turned out that the developer responsible for the module adopted *test-driven development*² methodology initially, but abandoned it during the development. This is very interesting. The telemetry streams in the chart clearly reveals the impact of the process-level changes³.

Telemetry streams consist of software process and product metrics. They are the basis of project management and process improvement. Telemetry charts and reports provide representation and display of telemetry trends. They make software developers more aware of their development processes by making them transparent and readily available. Telemetry streams for the same time interval are juxtaposed in the same telemetry chart or report to help developers detect covariance between different software metrics. For example, one might find that a drop in test coverage is frequently associated with an increase in the number of open bugs. This kind of information is important to 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

²*Test-driven development* is a programming technique emphasized in *extreme programming* [9]. It requires writing test cases first before implementing the actual code.

³Again, the telemetry analysis does not presume any assumption that *test-driven development* is better than other process methodologies. The judgment is left for the analysis user.

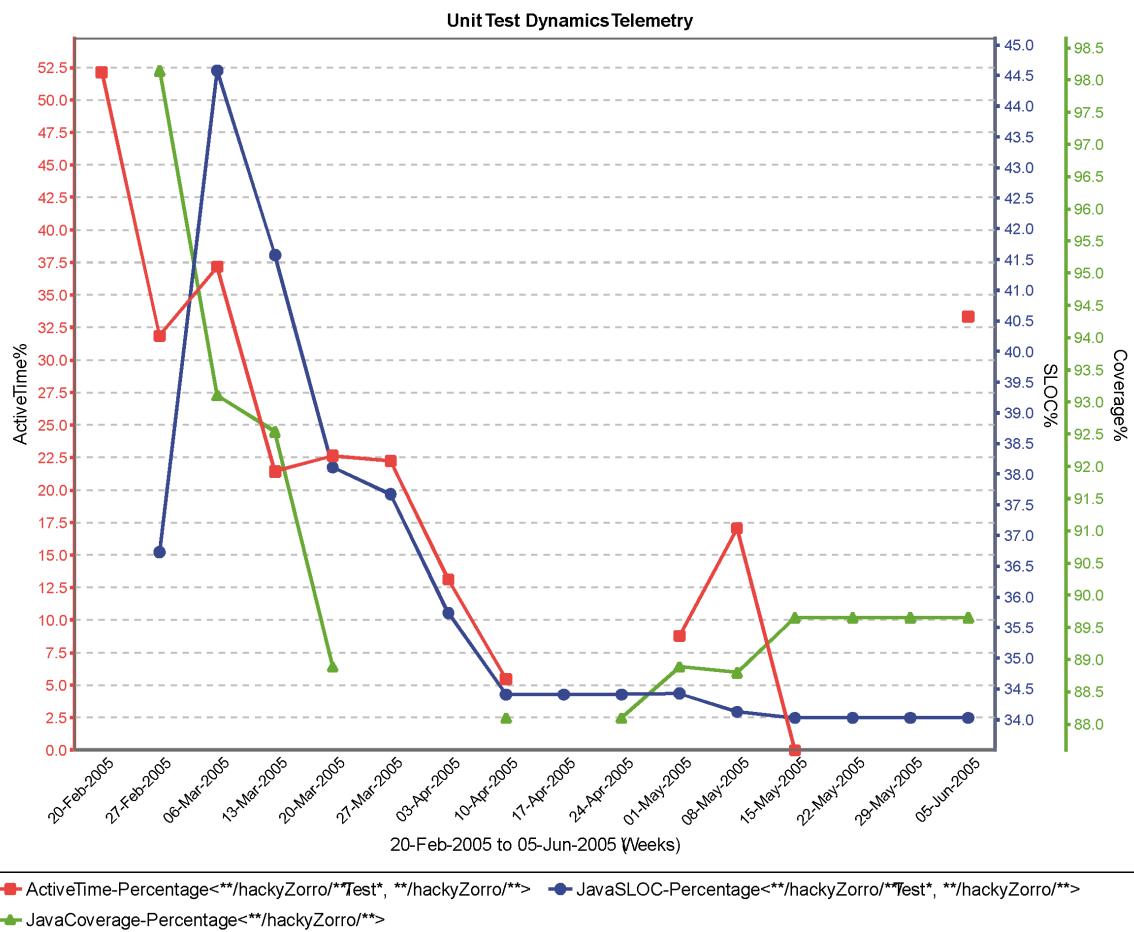


Figure 1.4. Telemetry Chart

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 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. The telemetry language provides a flexible mechanism decoupling the types of metrics and the types of analysis. It enables interactive exploration of the relationship between these different metrics.

In general, applications of software project telemetry involve the following cycles to empirically guide the decision-making for project management and process improvement:

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 plausible cause for the problem, and possible measure 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 claim of this thesis is that software project telemetry provides an effective approach to (1) automated metrics collection and analysis, and (2) in-process, empirically-guided software development process problem detection and diagnosis.

Compared to traditional model-base process prediction approaches, software project telemetry should be easier to use and cheaper to implement. It does not require software organizations to accumulate process and product metrics for finished projects in historical databases. 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 should be robust. The information contained in telemetry streams should seldom be affected when there is occasional metrics drop out, and analyses should still provide decision-making value even if metrics collection starts midway through a project.

Software project telemetry should also be flexible. There are no required set of metrics. Different software organizations can collect different sets 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 Empirical Evaluation

The claim of this thesis was evaluated in two empirical studies: one in a classroom setting, and the other in the Collaborative Software Development Lab (CSDL). The primary goal was to assess metrics collection cost and decision-making value of software project telemetry. The secondary goal was to discover obstacles the developers might encounter during their use of the technology, and to gain insights about software project telemetry best practices and possible technology adoption barriers.

The classroom study was conducted in the two software engineering classes taught by Dr. Philip Johnson at the University of Hawaii in Spring 2005: one class for senior-level undergraduate students, and the other for introductory-level graduate students. By curriculum design, the students were divided into groups of 2 - 4 members collaborating on group projects and introduced to use software project telemetry to collect metrics and perform analyses on their own data. There were 25 students participating the study: 9 from the undergraduate session, and 16 from the graduate session.

The CSDL study was conducted in the Collaborative Software Development Lab at the University of Hawaii in Spring 2006, when a large scale software system (i.e., the Hackystat system itself) was being developed and maintained by a team of five on-site developers and a project manager. Three of the developers were Ph.D. students in software engineering (including me). They were hired by the lab working 20 hours a week. The other two were undergraduate students in their final semester. They were top students from the undergraduate software engineering class. They were working for the lab in exchange for personal development and course credit.

The two software development environments were quite different. In the classroom, there were a relatively large number of developers working on small scale class projects. In CSDL, there were a relatively small number of developers collaborating on a much larger project, which contained almost 300,000 lines of code and had been under development for five years. The CSDL developers had significantly more software engineering experience and process maturity compared to the average student in the classroom.

As a result, the two studies were structured differently. The classroom study was “*passive*” in nature: though the students used software project telemetry to collect metrics and perform analyses on their own data, I did not make any deliberate attempts to help them improve their software development processes. On the other hand, the CSDL study was “*active*” in nature: I introduced software project telemetry as a metrics-based process improvement program; I helped the project manager institute changes to improve project management practices; I also helped the developers gain insights into their development process. Different data collection and analysis techniques were used in the two studies. The classroom study was relatively simple. My goal was to gather insights from a relatively large number of developers in a relatively short period of time. I distributed a questionnaire at the end of the semester to collect the student’s opinions about software project telemetry. To increase my confidence in the validity of their self-reported opinions, I also analyzed their telemetry analysis invocation pattern to determine the extent to which their opinions were based on the actual system usage. In the CSDL study, I pursued a much more in-depth data collection and analysis strategy over a much longer period of time. I collected data from observations and interviews; I generated hypotheses from the data; I also tested the hypotheses in a limited way by making changes to the telemetry system or implementing new facilities to see whether the hypothesized outcome would come true or not.

The results of the two studies suggested that software project telemetry had acceptably-low metrics collection and analysis cost, and that it provided project management and process improvement decision-making values.

1.7 Contribution

There are three main contributions from this research:

1. The Concept of Software Project Telemetry

In metrics collection, software project telemetry uses sensors to collect metrics automatically and unobtrusively. This sensor-based approach eliminates the chronic “context-switch” overhead inherent in manual approaches, such as PSP, and tool-assisted approaches, such as LEAP, PSP Studio, and Software Process Dashboard.

In metrics decision-making, software project telemetry follows a light-weight approach by comparing telemetry trends in two different periods from the same project. The comparison involves a much smaller time scale than the whole project lifecycle. The metrics from the initial period of the project are used to establish a baseline and bootstrap the process. Project management and process improvement decisions are made by detecting changes in telemetry trends and comparing trends in two different periods in the same project. In-process control for a project that is still under development is made possible precisely because comparisons are made within a project. Since this approach does not involve the building of a statistical model in order to make cross-project comparison, it avoids many problems that typically exist in model-based approaches, such as spending the cost to accumulate a historical database of projects that may not be “comparable” to the current project.

The two empirical studies I conducted showed that software project telemetry had sufficiently low metrics collection and analysis cost, and that it was able to deliver decision-making values, at least within the exploratory context of the two studies.

2. The Implementation of Software Project Telemetry

Two pieces of software are the direct result of this thesis research. One of them is a server-side component, which includes the software code to interpret the telemetry language, the code to perform telemetry analyses and generate telemetry charts, and a web-based management

console for telemetry construct definitions. The server-side component enables a user to log on to the server through a web browser to “*actively*” explore relationships between different software metrics.

The other is a client-side application that can be configured to automatically retrieve telemetry charts from the server and display them. It makes a sequence of telemetry charts continuously available to the user, providing “*passive*” awareness of project status.

The source code is GPL licensed, which encourages third-party improvement to be contributed back to the community. The system has already been adopted by several external sites, such as Sun Microsystem and the University of Maryland.

3. The Insights from the Empirical Studies

Software project telemetry delivers best decision-making value when it can be customized to the specific needs of a software organization. The customization includes both setting up sensors to collect metrics and designing telemetry charts to perform analyses. “Top-down telemetry design” and “bottom-up metrics collection” are best practices. Top-down telemetry design refers to the idea that each telemetry chart should be designed with a clear purpose in mind, such as to help the development team meet a specific improvement goal. Bottom-up metrics collection refers to the recommendation to collect whatever metrics a software organization can. The rationale is the low cost associated with sensor-based metrics collection. Even if there is no apparent need for a metric today, it can still be used to establish a baseline for comparison tomorrow.

Due to the automated nature of metrics collection in software project telemetry, broken sensors might not be noticed immediately. However, the empirical studies suggested that it would be possible to design special-purpose telemetry charts to help developers make quick assessments of whether the underlying sensors are sending data correctly or not. Therefore, another best practice for an organization is to deploy these special-purpose charts and assign a designated person to examine them.

One adoption barrier for this technology is concern about the level of privacy and confidentiality accorded to the data, especially with the effort-related personal process metrics. Though the current implementation has a mechanism to limit the kinds of data that could be accessed by people other than the owner, overcoming this issue seems largely dependent on what the data are used for in an organization. In other words, are the data used to improve development processes or to evaluate developers’ performance? Lack of telemetry expertise within

an organization might be another technology adoption barrier, since software project telemetry will not likely deliver the best value if used straight “out of the box” and effective use, at least at this point, appears to require customization of the telemetry charts and reports.

1.8 Thesis Organization

This thesis is organized into the following chapters:

- Chapter 1 is this chapter where the problem statement and proposed solution is described.
- Chapter 2 relates the current research to the broader context of existing work.
- Chapter 3 describes software project telemetry in detail.
- Chapter 4 describes the design details of an 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 case study of software project telemetry in software engineering classes.
- Chapter 7 reports on a case study of software project telemetry in the Collaborative Software Development Lab at University of Hawaii.
- Chapter 8 synthesizes the results from the two case studies to gain further insights.
- Chapter 9 provides final concluding remarks of this thesis research.

Chapter 2

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 the trends 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 metrics-based approaches. Some approaches, such as the Personal Software Process (PSP) [42, 43], 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) [8, 7], and the Software Capability Maturity Model (CMM) [66, 74], are only comparable with respect to process methodology.

This chapter proceeds with an overview of software measurement theory in Section 2.1, which serve as the foundation for any software measurement programs. Section 2.2 discusses the Personal Software Process. Section 2.3 discusses process prediction models, especially the Constructive Cost

Table 2.1. Software Measurement Classification

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

Model. Section 2.4 discusses the Goal-Question-Metric paradigm. Section 2.5 discusses maturity frameworks, especially the Software Capability Maturity Model. Section 2.6 concludes the chapter with a summary.

2.1 Software Measurement Theory

As noted by DeMarco [22]: “*You can neither predict nor control what you cannot measure.*” Measurement is the first step to transform software engineering from an art where the success of a project depends largely on the competence and commitment of individual developer, to a scientific discipline where project outcome is both predictable and controllable.

Measurement is defined as 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 [29]. 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 two categories: software product and process. An attribute is a property of an entity, such as the size of a program, the size of test scripts, and the time required to finish a milestone. Attributes are generally divided into two categories: internal and external. Measures for internal attributes can be computed based on the entity itself; while measures for external attributes depend on both the entity and the environment in which the entity resides. The resulting classification scheme is depicted in Table 2.1. The cell for internal software process attributes is empty, because all software process metrics are dependent on the environment to various degrees.

The representation theory of measurement formalizes the process of mapping from an empirical relation system to a numerical relation system. A “measure” is nothing but the number assigned to describe some attribute of an entity by the mapping process. 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 numbers 1, 2, and 3, respectively. Since sports competition results

Table 2.2. Measurement Scale and Valid Mathematical Operations

Measurement Scale	Valid Mathematical Operators
Nominal Scale	=
Ordinal Scale	= > <
Interval Scale	= > < + -
Ratio Scale	= > < + - * /

contain 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 a numerical relation system are valid in the original empirical relation system.

Formally, the mapping, together with the associated empirical and numerical relation systems, is called the “*measurement scale*.” It is the measurement scale that determines valid mathematical operations that can be performed. In general, measurement scales are classified into four categories with increasing level of restrictiveness: nominal, ordinal, interval, and ratio. More restrictiveness means more mathematical operators in the numerical relation system can be applied. Table 2.2 lists the types of measurement scale and their corresponding valid mathematical operators.

- **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 number 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 known 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 line,” “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 computation 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. Telemetry reducers implement different reduction behaviors. They form the lowest level, atomic “building blocks” of the software project telemetry observable by an end user. Though data points in telemetry streams are mapped to real numbers by the reduction process, they can be of any measurement scale in theory. The reduction process itself is treated as a black box by the telemetry infrastructure. This is not a problem to end users, because the internal implementation details of telemetry reducers are not exposed to them. However, the developers who are responsible for reducer implementation must make sure that sensor data are manipulated in a meaningful way.

Telemetry language processing acts on telemetry streams. It includes telemetry function calls and telemetry arithmetic operations. The data points in telemetry streams are treated as if they were of ratio scale by the language interpreter. As a result, the language allows addition, subtraction, multiplication, and division between telemetry streams. This might cause a problem to a careless user, because there is the theoretical possibility of scale type mismatch. In other words, the telemetry language might allow meaningless mathematical operations to be applied to different types of metrics, such as adding code churn metric to unit test coverage metric. The problem could be solved by introducing a “type” system to the language, but doing so would significantly complicate the language design and its implementation. Currently, software project telemetry takes a pragmatic approach by relying on the user defining telemetry constructs to recognize nonsense operations. A topic for future research is to determine whether scale type mismatch is a significant problem in the use of software project telemetry, and to devise appropriate mechanisms to detect and/or prevent this problem.

2.2 Personal Software Process

The Personal Software Process (PSP)¹ [42, 43] is a self-improvement process for software developers, and a ground-breaking approach that adapts organizational-level software measurement and analysis techniques to individuals.

The PSP provides both measurement machinery and process methodology. The primary goal of the PSP is to improve project estimation and quality assurance. The goal is pursued by 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 make 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 development process.

The original PSP proposed by Humphrey uses the very tedious manual approach to collect and analyze metrics. For instance, every time a compilation error occurs, the developer has to stop his/her current work, and log on paper forms the details of the error. Though several studies [30, 37, 56] 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 [52] 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” [42] that “*it would be nice to have a tool to automatically gather the PSP data.*”

Tools, such as LEAP [65], PSP Studio [38] and Software Process Dashboard [82], do 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

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

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 [51, 53]. This chronic context switch appears to be a problem for many 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 collect metrics automatically and unobtrusively so that developers are not distracted from their primary tasks – developing software products. Compared to manual and tool-based metrics collection, the 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, along with its restrictions, are discussed in Section 3.2.

With respect to process methodology, the PSP uses observation-evaluation-modification cycles to improve software development process. One cycle corresponds to the life time of a 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, which requires a sufficient number of data points with respect to time and size of the 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 in Section 2.3, since all model-based approaches face the same limitation.

Similar to the PSP, software project telemetry uses cycles to improve the software development process. The cycle includes process problem detection, hypothesis generation, change implementation, and hypothesis validation. The difference is that a software project telemetry cycle does not correspond to the life time of a project. It involves much smaller time scale, and a single project typically has many cycles. The idea is that comparisons can be made between two different periods of the same project instead of between two different projects, and that the changes in the development

²Sensor-based approach to metrics collection is pioneered in the Hackystat project [53], developed in the Collaborative Software Development Lab at University of Hawaii. I have been on the Hackystat development team since 2002 while doing software project telemetry research.

process and their trends can be used as the basis for decision-making in project management and process improvement. Since software project telemetry does not make model-based cross-project comparisons, there is neither a need to accumulate a historical project database, nor a necessity to ensure context consistency between different projects.

2.3 Constructive Cost Model and Model-based Process Predictions

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 into the details of the 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 model building. 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 (i.e., 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 tends 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 follows similar approaches in model building and process prediction, COCOMO is used to illustrate how they work in general. COCOMO is chosen because it is one of the most widely available and accepted models in the public domain. It 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. COCOMO 81 was the original model published in the book *Software Engineering Economics* [10]. It offers three 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 three levels: application composition, early design, and post-architecture, to explicitly model the fact that uncertainty of effort and schedule estimates decreases through software project life cycle.

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

$$PM = A * \prod_{i=1}^{17} EM_i * Size^{(B+0.01*\sum_{i=1}^5 SF_i)} \quad (2.3.1)$$

$$TDEV = C * PM^{(D+0.002*\sum_{i=1}^5 SF_i)} \quad (2.3.2)$$

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 convert other counting results, such as function points⁴ [2] and object points [4, 5], 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: *Very Low, Low, Nominal, High, etc.*

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

⁴A 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 point 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, the 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, the calibration values for the COCOMO II 2000 post-architecture model 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) [14], COCOTS (Constructive COTS Model) [1], and CORADMO (Constructive Rapid Application Development Model) [28]. Commercial implementations include Costar from Softstart Systems [18], Cost Xpert from Cost Xpert Group Incorporated [84], 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 in practice.

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 [67], 67% of them are at CMM 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 are incapable of benefiting from model-based process prediction techniques, such as COCOMO.

Second, the prediction power of these models is highly dependent 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 the current project may differ from those in the historical project database, not to mention the practicality and cost of accumulating such a historical project database in the first place.

Lastly, model-based process predictions are primarily designed to be used at a 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. The project manager will know the answer after the entire project is finished, but by that time the information is irrelevant. To put it simply, end-point estimation is not very effective for in-process control.

Software project telemetry avoids the above-mentioned difficulties by shifting the focus of process prediction. It makes no attempt to build a 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 changes in software development process and the trends of those changes to make short-term predictions for the purpose of in-process project management.

2.4 Goal-Question-Metric Paradigm

The Goal-Question-Metric paradigm (GQM) [8, 7] provides a top-down, goal-oriented process methodology: software measurement is driven by high level goals. Usually the business goals of an organization are formed first, and then translated into improvement goals of software development, which, in turn, are translated into measurement goals. A metrics program is used to fulfill these measurement goals. Based on the measurement results, the organization can generate hypotheses and make decisions to reach the software development improvement goals, and, finally, the business goals.

GQM measurement goal is stated in 5 dimensions: *study object, purpose, quality focus, view point, and environment*. A concrete example can be found in [76, 75], in which the authors studied causes and effects of interruptions 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.

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

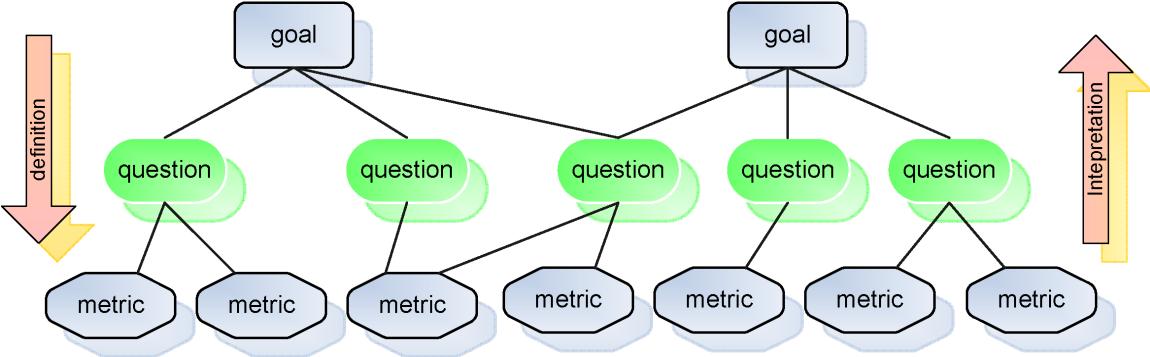


Figure 2.1. Goal-Question-Metric Paradigm

GQM includes a top-down methodology that translates measurement goals into questions that need to be answered in order to reach them. Based on the questions, a set of relevant metrics can be identified and collected, which provide answers to the questions. The methodology can be best visualized as a tree-structured graph in Figure 2.1.

The GQM paradigm is well-known, and case studies of successful experience abound [6, 60, 32]. The key to a success 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. However, the main criticism of GQM is the lack of attention to the actual measurement process: metrics collection and interpretation are not part of the paradigm. GQM implicitly assumes that once all required metrics are identified, the rest of the steps (metrics collection and interpretation) are easy.

Software project telemetry and the GQM paradigm can complement each other: GQM defines useful software metrics and relates them to an organization’s business, project, product, and process goals; while software project telemetry provides an automated machinery for metrics collection and analysis. For example, “Continuous GQM” [62] tries to implement GQM in an automated way in which data is collected, analyzed, and presented automatically with minimal human effort .

2.5 Capability Maturity Model and Process Maturity Frameworks

The Capability Maturity Model (CMM) [66, 74] is a process maturity framework developed by the Software Engineering Institute (SEI). Other similar frameworks include ISO 9001 [45], SPICE,

Table 2.3. Process Maturity and Measurement

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

and ISO/IEC 15504 [46, 27]. These frameworks share common properties, such as using metrics as a 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 a 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 the 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. Process maturity frameworks offer a convenient context to plan software project telemetry program so that it grows to embrace additional aspects of software 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 level process maturity where quantitative measurement is required. Indeed, several authors have studied the relationship between software measurement and process maturity. For example, Table 2.3 lists Pfleeger and McGowan’s recommendation of collecting different measures depending on a software organization’s CMM maturity level [68].

The following discussion uses CMM as an example to illustrate how a maturity framework works. Note that CMM is used here as a generic term referencing the works published by SEI, which include SW-CMM v1.0 (1991), SW-CMM v1.1 (1993), CMMI v1.02 (2000), and CMMI v1.1 (2002). CMMI [73] is the acronym for Capability Maturity Model Integration.

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 the 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. 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.

Table 2.4. CMM Levels and Key Process Areas

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

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 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 2.4 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 [44, 24, 21, 23]. For example, Humphrey *et. al.* [44] 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 how the measurement process itself should be implemented. In fact, Humphrey appears to be aware of the difficulty in implementing a measurement program in a software organization. He mentioned in [74] that:

“The greatest potential problem with the managed process 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.”

Software project telemetry can complement CMM. It provides not only an automatic and unobtrusive way of gathering software metrics 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 for software organizations to achieve CMM Level 4 and 5.

2.6 Chapter Summary

In this chapter, I have compared and contrasted Software Project Telemetry to other metrics-based approaches. The Personal Software Process suffers from chronic metrics collection and analysis overhead. The Constructive Cost Model relies on the assumption that software development process is stable and thus predictable. The Goal-Question-Metric paradigm offers a high-level abstract process methodology but lacks attention to the actual measurement process. The Capability Maturity Model prescribes that measurement should be taken to assess the status of development activities but does not specify how software measurement itself should be implemented. Software Project Telemetry addresses these limitations through automated metrics collection and analysis, and in-process, empirically-guided software development process problem detection and diagnosis. The next chapter gives a detailed discussion of Software Project Telemetry.

Chapter 3

Software Project Telemetry

Software project telemetry is a novel light-weight software 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 to telemetry streams, charts, and reports through a domain-specific language for the representation of telemetry trends for high-level perspectives on software development processes. Compared to traditional metrics-based approaches, which are based primarily on historical project databases and model-based comparison, software project telemetry emphasizes project dynamics and in-process control. The comparison in software project telemetry involves much smaller time scales. The idea is that comparison can be made between two different periods of the same project instead of between two different projects, and that the changes in the development process and their trends can be used as the basis for decision-making in project management and process improvement.

This chapter is organized into the following sections. Section 3.1 gives an overview of software project telemetry and its essential characteristics. Section 3.2 discusses sensor-based metrics collection. Section 3.3 discusses telemetry language and telemetry constructs such as stream, chart, and report. Section 3.4 discusses the telemetry-based methodology for project management and process improvement. Section 3.5 summarizes this chapter.

3.1 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 starting from the early Gemini missions to the modern Mars rovers. 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.

The same concept can be applied to software project management: software project telemetry is an automated process and product measurement approach. It is used to gain insight into software development processes, detect early signals of project failures, and analyze impact of project decisions. In this approach, sensors unobtrusively collect time-stamped software metrics, which are abstracted into telemetry streams, charts, and reports. Trends in telemetry streams serve as the basis for project management and process improvement. Telemetry charts and reports provide visualization. By detecting changes and covariance in trends of different metrics, software project telemetry enables a more incremental, visible, and experiential approach to project decision-making. It has the following essential characteristics:

1. Software project telemetry data are collected automatically by sensors 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 [42].
2. Software project telemetry data consist of a stream of time-stamped events, where the timestamp is significant for analysis. Software project telemetry is thus focused on evolutionary process in software development. This contrasts, for example, with COCOMO [10, 11], where the time at which the calibration data are collected about the project is not significant.
3. Software project telemetry data are continuously updated and immediately available to both developers and managers. Telemetry data are not hidden away in some obscure database

guarded by the software quality improvement group. They are easily visible to all members of the project for interpretation.

4. Software project telemetry exhibits graceful degradation. While complete sensor data provide best support for project management, telemetry analyses still provide decision-making value even if there is occasional dropout of sensor data, or 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.

3.2 Sensor-based Data Collection

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

Software process metrics are the metrics that assist in monitoring and controlling the way software is produced. 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 [78], and Eclipse [26]. 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 Clear Case [15], CVS [20], and SVN [79]. 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 [49]. Whenever an issue is reported or its status is updated, the sensor can detect such activities and record the relevant information.
- A plug-in for an automated build system, such as Cruise Control [17]. It can capture information related to build attempts and build results.

Software product metrics are the metrics that describe the properties of the software itself. Sensors collecting product metrics are typically implemented as analyzers for software artifacts. These analyzers usually 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* job¹, or run them as tasks in automated build system. Some examples are listed below:

- An analyzer that parses program source code to compute size or complexity information.
- An analyzer that parses the output of existing tools, such as Clover [16], and JBlanket [48], and converts them to a data format that can be used by software project telemetry.

There are many other possibilities. One can even imagine an exotic sensor that retrieves project cost and payroll information from a company’s accounting database, if extraction of such information is permitted by the company policy. The point is: no matter what the sensor does and regardless of its implementation details, a sensor-based approach collects metrics *automatically* and *unobtrusively* in order to keep data collection cost low, so that developers are not distracted from their primary tasks – developing software products instead of capturing process and product metrics.

This sensor-based approach eliminates the chronic overhead in metrics collection. While setting up sensors might require some effort, once they are installed and configured, sensor data collection is automatic. This contrasts with traditional data collection techniques, such as the paper-and-pencil based approach used in PSP/TSP [42], or the tool-supported approach used in LEAP [65], PSP Studio [38], and Software Process Dashboard [82]. These approaches require constant human intervention or developer effort to collect metrics. Even in the case of the tool-supported approach, the developer still cannot escape the chronic overhead of constantly switching back and forth between doing work and telling the tool what work is being done [51, 53].

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

Admittedly, the sensor-based approach does come with 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 25 sensors for commonly-used tools.
- Some metrics may not be amenable to automated data collection. An example is software development effort. While it is feasible 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 almost impossible to collect this type of effort in an automated way. It is still an open research question whether *all* important metrics can be captured by sensors or not. However, this research takes a more pragmatic view: it is only concerned with whether sensors can collect *sufficient* metrics so that software project telemetry has decision-making value for project management and process improvement.

3.3 Telemetry Language and Telemetry Constructs

Many interesting issues in software project management involve understanding the relationship between different measures. For example, we might be interested in seeing whether 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 questions require comparing a set of metrics values over time. The telemetry language provides a mechanism that facilitates interactive exploration of relationships between metrics. The language has the following syntax:

```
streams <StreamName> (<ParameterList>) = {  
    <DocumentationString>,
```

```

<Expression>
};

y-axis <YAxisName> (<Parameter>) = {
    label, 'integer|double|auto', lowerBound, upperBound
};

chart <ChartName> (<ParameterList>) = {
    <ChartTitle>,
    <StreamReferences>
};

report <ReportName> (<ParameterList>) = {
    <ReportTitle>,
    <ChartReferences>
};

```

Note, however, that the above syntax is not written in a strict mathematical notation. The formal grammar of the language can be found in Appendix A: *Software Project Telemetry Language Specification*. Chapter 1 contains a real-world example of the language in Section 1, and the resulting telemetry report in Figure 1.2.

Telemetry *reports*, *charts*, and *streams* are basic constructs of the language. The relationship between them is illustrated in Figure 1.2 and discussed in Section 1.4.2. In essence, 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 of one or more process or product metrics over time. The *y-axis* construct is used to specify the vertical axis of a telemetry chart. Note, however, that a telemetry chart definition does not include the information about its horizontal axis, because such information can be automatically inferred from the time interval over which the telemetry analysis is performed. A telemetry stream is a sequence of a single type of software process or product metrics.

The data collected by sensors are time-stamped, and the time stamp is always significant in telemetry style metrics analysis. There may not be a simple one-to-one correspondence between sensor data and the data points in telemetry streams. Sensor data usually represents very fine-grained

low level software process or product details, while the data points in telemetry streams represent higher level perspectives on the software system or its development process. Typically, sensor data need to be filtered, combined, and aggregated to derive a telemetry data point. For example, suppose we want to construct a telemetry stream representing the number of open bugs for a software project on a monthly interval for the entire year 2005. In addition, suppose we are using Bugzilla [13], and 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 2005, the telemetry reducer needs to scan the entire bug event sensor data and combine them in order to compute values for the telemetry data points.

Telemetry reducers takes sensor data as input and output a series of telemetry data points. They are the “atomic” building blocks for telemetry constructs at the user level. They serve as the link between sensor data and telemetry streams. They can be thought of as the fixed “alphabet” from which any number of telemetry streams can be created by a user. A reducer must be available in order to construct a “*simple*” telemetry stream². Some example of general classes of telemetry streams 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 [20], Clear Case [15]), issue management systems(e.g., Bugzilla [13], Jira [49]), 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 [17]), testing tools(e.g., JUnit [55]), size and complexity counters(e.g., LOCC [61]), etc.
- **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,

²Telemetry streams can also be generated by applying mathematical operations or telemetry functions to existing telemetry streams. These are called “*compound*” telemetry streams as opposed to “*simple*” telemetry streams.

occurrence of exceptions), or to load testing tools (e.g., JMeter [50]) 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.

3.4 Telemetry-guided Project Management and Process Improvement

The basic steps of telemetry-guided process improvement are illustrated in Figure 3.1. It involves cycles of problem detection, process improvement hypothesis generation, process change implementation, hypothesis validation, and impact analysis. Following Hetzel [39], 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 generates a basic set of telemetry streams, which provide insights into the current software development practice and help establish a base line for the current process.

Software project telemetry can be used in two modes: (1) *in-process project monitoring*, and (2) *process improvement*. The two modes are closely related, and sometimes indistinguishable in practice. However, I will keep them separated in this discussion in order to make the concept clear.

The steps for in-process project monitoring start from the upper arrow in Figure 3.1. 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 the development process, can be used to detect correlations. For example, the project manager 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 might help the project manager generate hypothesis about the causal relationship and corrective measures for process improvement. For example, the project manager might identify code complexity as a likely cause for high defect density. Once the process improvement hypothesis is generated, the project manager tries corrective actions 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. The project manager 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

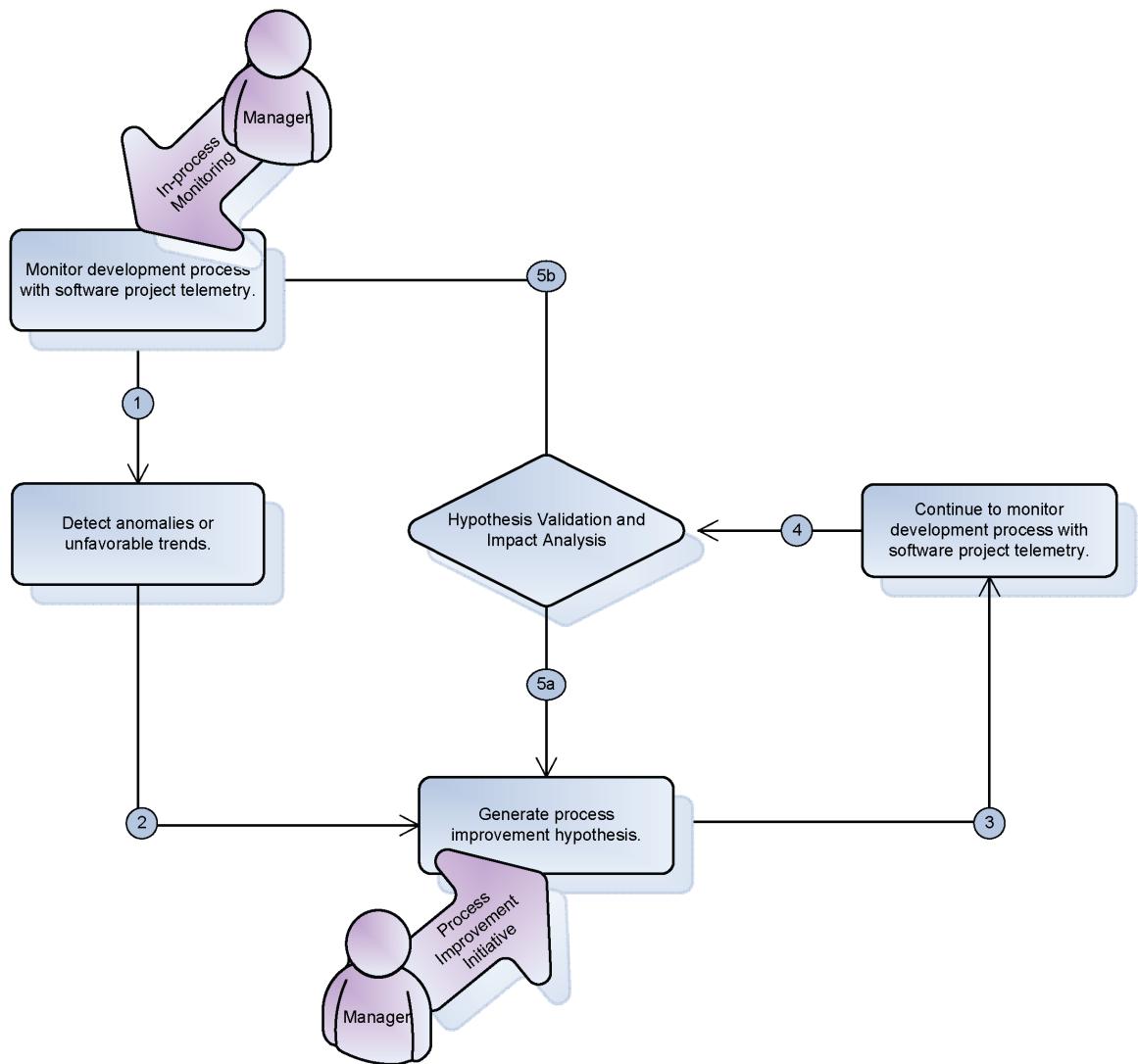


Figure 3.1. Telemetry-based Process Improvement

be validated by the newly-generated telemetry streams. If the 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 steps for process improvement follow the same loop as the steps for in-process project monitoring. The only difference is that they start from the lower arrow in Figure 3.1, and implementation of process improvement measures does not have to wait until anomalies or unfavorable trends are detected in telemetry streams.

3.5 Chapter Summary

In this chapter, I have described Software Project Telemetry in theory. 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. The next chapter introduces the tool implementation.

Chapter 4

Implementation

My implementation of software project telemetry has been integrated into the Hackystat system. Hackystat is an open-source framework developed in the Collaborative Software Development Lab (CSDL) at the University of Hawaii for automated collection and analysis of software product and process metrics and empirical software engineering experimentation. I have been contributing to its development since 2002. The relationship between my implementation of software project telemetry and Hackystat is illustrated in Figure 4.1. The implementation can be viewed as an extension to the Hackystat framework. CSDL members often use “Hackystat” as an umbrella term to refer to the framework plus all the extensions built on top of it. However, for the purpose of clarity, I will try to make a distinction between them. Throughout this chapter:

- The “*Hackystat framework*” refers to the core framework components that provide metrics storage service and extension plug-in mechanism.
- The “*software project telemetry*” refers my implementation of software project telemetry.
- The “*Hackystat system*” refers to the framework plus all the extensions built on top of it.

This chapter starts with a brief overview of the Hackystat framework and the services it provides in Section 4.1. Section 4.2 describes my implementation of software project telemetry, which utilizes Hackystat framework services. Section 4.3 introduces all available telemetry *reducers* and *functions*. Section 4.4 summarizes this chapter.

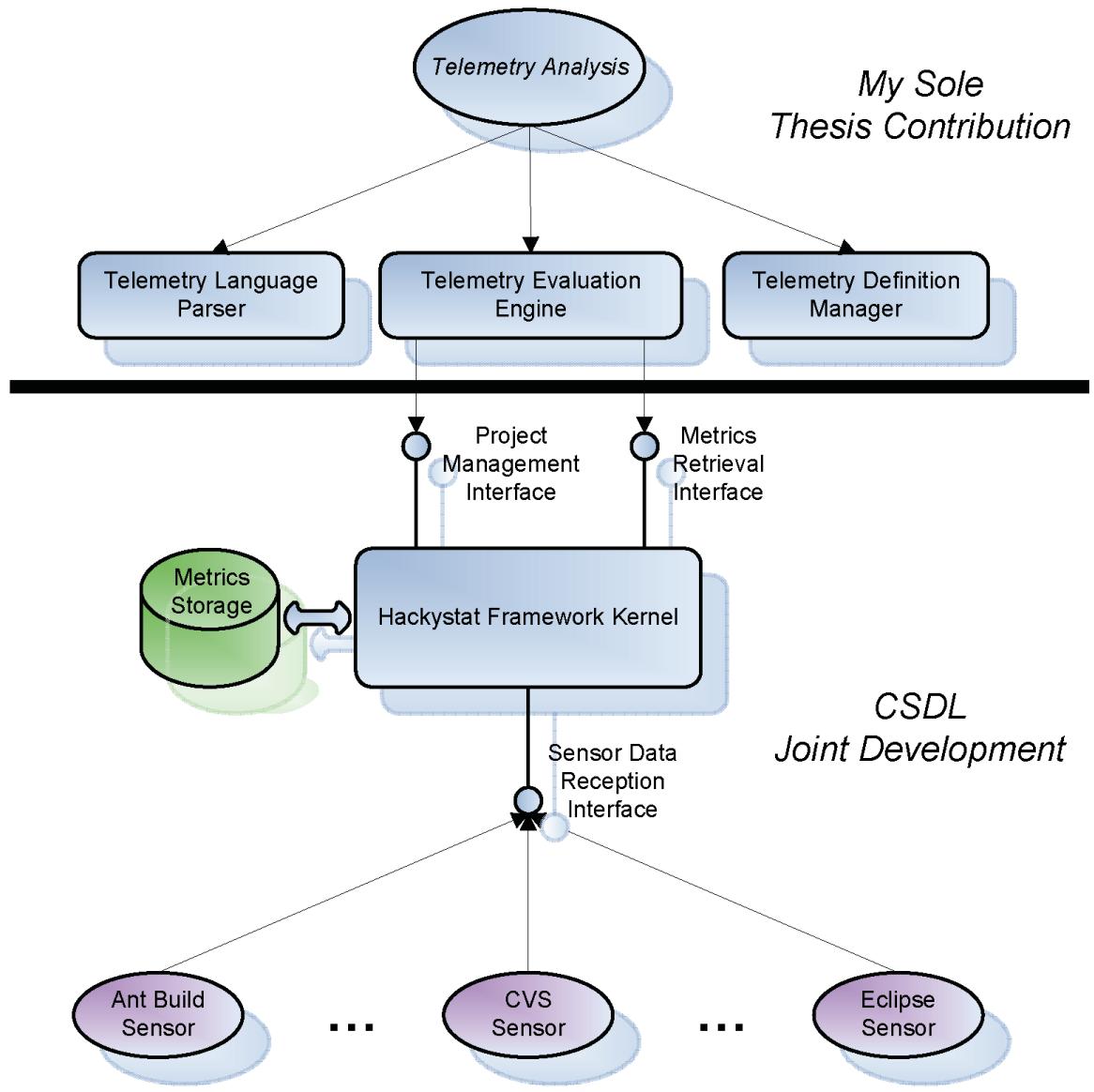


Figure 4.1. Software Project Telemetry System Implementation

4.1 Hackystat Framework

Hackystat is an open-source framework developed in the Collaborative Software Development Lab (CSDL) at the University of Hawaii for automated collection and analysis of software product and process metrics and empirical software engineering experimentation. The framework itself is tool, environment, process, and application independent. It does not presume a specific operating system platform, a specific integrated development environment, a specific software process, or a specific application area. It is designed to be extended. Therefore, it only provides generic services such as metrics storage, project definition management, and an extension mechanism where new modules (functionalities) can be plugged in.

A Hackystat system is configured from a set of modules, which determine the actual functionality of the system: which development tools are supported, what types of software metrics are collected, and what analyses are run on the metrics. For example, several different research projects are being conducted using Hackystat. One research involves a Hackystat system configured from a set of modules specialized in low-level software process analysis [57]. Another research involves another Hackystat system configured from another set of modules trying to understand parallel software development for high performance computers [54]. This thesis reports on my research involving yet another Hackystat system configured from yet another set of modules supporting project management and process improvement.

4.1.1 Metrics Storage

The Hackystat framework exposes two interfaces related to metrics storage as illustrated in Figure 4.1: one for metrics data reception, and the other for metrics retrieval.

The metrics data reception interface is used by sensors to send software process and product metrics. The communication occurs on an HTTP SOAP channel. Hackystat developers often refer to the Hackystat architecture as a client-server system. In this view, the “*clients*” are development environment tools, such as editors (Emacs, Eclipse, Vim), configuration management systems (CVS, Harvest), build tools (Ant, Make), unit testing tools (JUnit), and so forth. For each of these

tools, a custom sensor must be developed. The “*server*” refers to the Hackystat framework kernel plus all the analysis extensions. They run inside a standard J2EE¹ application container.

The metrics retrieval interface is used by metrics analysis extensions on the server side. Strictly speaking, the analysis code might opt not to use this interface directly. Instead, it may choose to rely on a higher level metrics abstraction mechanism which in turn depends on the metrics retrieval interface.

The Hackystat framework kernel handles metrics data storage automatically. The persistence engine is completely opaque, visible neither to the client-side sensors nor to the server-side analysis code. The current kernel implementation stores all metrics data in plain file system files in XML format.

4.1.2 Project Definition Management

Sensors simply send bits of raw data concerning software process or product to the server. They know nothing about the larger context in which the development is performed. For example, during a typical day, a developer might work on several distinct tasks: an hour in the morning on requirements for an upcoming project, and two hours in the afternoon on maintenance fixes to an old system. For the requirements project, the developer is working in a team with one other person; while the system maintenance and development involves 12 people. In most cases, the developer will want to analyze the requirements data separately from the maintenance data. The developer will also probably want to gain a higher level perspective on the progress of the requirements project by combining his data and the relevant data from the other person he is working with. Similarly, it would be helpful to combine together the relevant data from all the 12 people working on maintenance to see how that project is progressing.

The Hackystat framework kernel supports team level analysis through project definitions. The project definition is designed to specify a context for analysis of sensor data, including the set of workspaces containing the artifacts associated with the project, the set of Hackystat users who are participating in the project and whose metrics should be combined together, and the time period during which the project is underway.

¹J2EE stands for Java 2 Platform Enterprise Edition. Starting from version 5, Sun Microsystem has re-branded it as Java EE.

4.1.3 Extension Mechanism

A Hackystat system provides all end user functionalities through extension modules. The framework exposes “*extension points*” to support extensions along multiple dimensions including sensors, metrics types, metrics analysis, documentation, and so forth. For each new functionality, the framework requires the developer to specify a declaration file in XML format to supply information about the specific extension implemented. Detailed information about the extension points is available in the Hackystat developer documentation, which can be found at the Hackystat home page: www.hackystat.org.

4.2 Hackystat Telemetry Module

The core of my implementation of software project telemetry resides in a Hackystat extension module called “*Core_Telemetry*”. It contains about 15,000 lines of code. The module itself is extensible to accommodate new metrics types and analyses that might arise in the future. It exposes two extension points of its own, where custom implementation of *telemetry reducers* and *telemetry functions* can be plugged in.

Just like the functionality of a Hackystat system is determined by its constituent modules, the functionality of the telemetry module hinges on the availability of reducers and functions. I have implemented over a dozen reducers and several functions to support this thesis research. They are distributed in the Hackystat modules where different types of software metrics are defined. They constitute about 13,000 lines of code. Available reducers and functions are listed in Section 4.3.

4.2.1 Functional Description

The software project telemetry implementation provides three analyses where a user can perform telemetry related exploration.

- **Telemetry Expert Analysis** — Figure 1.3 is a screenshot of the telemetry expert analysis. The user uses telemetry language to interact with the system directly to explore trends and relationships between different software product and process metrics. This is the most powerful and flexible analysis, and the user has the finest control. The expert analysis is especially

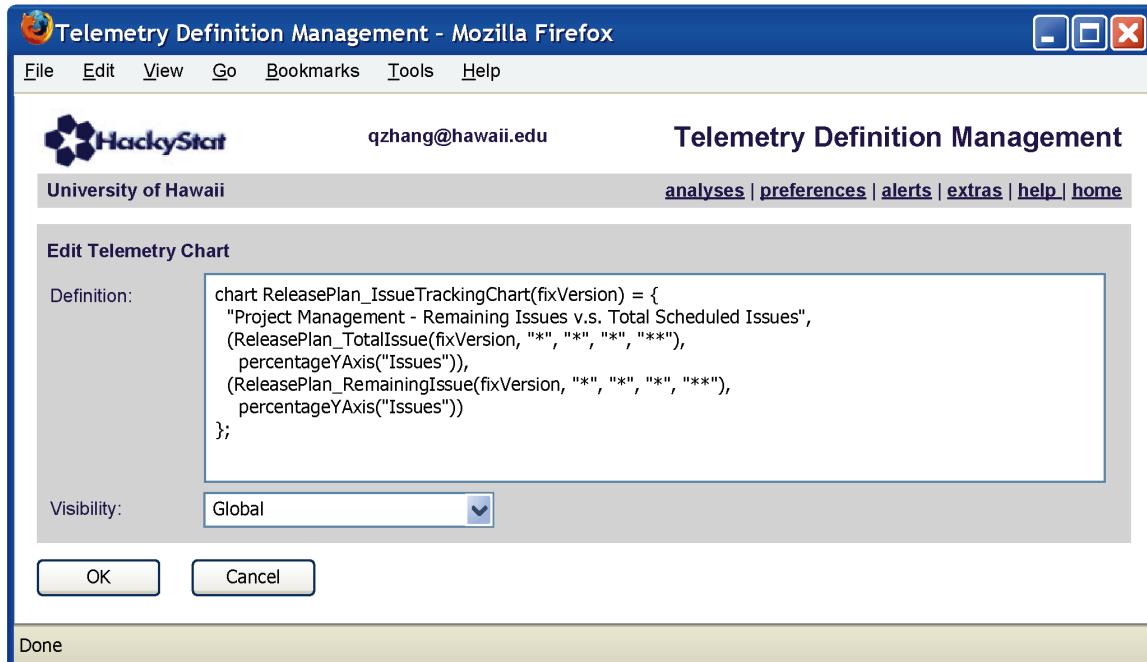


Figure 4.2. Telemetry Definition Management Console

useful when a user is experimenting with different forms of telemetry streams. Once the user is satisfied with the generated chart or report, he can create a persistent definition through the telemetry definition management console, so that later invocation of the same analysis can be performed through the telemetry chart/report analysis saving the effort of typing the definition again. Figure 4.2 is a screen shot showing a user creating a persistent definition of a telemetry chart with the definition management console.

- **Telemetry Report Analysis** — Figure 1.2 is a screenshot of the telemetry report analysis. This analysis performs the same analysis as the telemetry expert analysis, except that it hides the telemetry language from the end user. Instead of typing telemetry definitions directly, a user selects a predefined telemetry report from a drop-down list and performs the analysis. The telemetry definition management console shown in Figure 4.2 allows a system administrator or a telemetry expert to define a set of commonly used telemetry charts and reports and made them available in the drop-down box that the end user sees. This is a good way to lower the adoption barrier of software project telemetry, because it eliminates the need for a normal user to learn the telemetry language.

- **Telemetry Chart Analysis** — The telemetry chart analysis is similar to the telemetry report analysis. The only different is that the report analysis generates a group of related charts, while the chart analysis generates one single chart.

4.2.2 Implementation Details

The core of software project telemetry implementation resides in a Hackystat extension module called “*Core_Telemetry*”. The package structure is illustrated in Figure 4.3. It includes the following components:

- **Telemetry Language Parser** — The telemetry language parser parses user input of telemetry definitions into an abstract syntax tree, which is a Java object representation of telemetry definitions. The formal grammar of the telemetry language is specified in Appendix A. Figure 4.4 is a UML diagram for static structure of the telemetry abstract syntax tree. Internally, the parser is generated using JavaCC [47]: a top-down Java parser generator.
- **Telemetry Streams Data Model** — This is the data model for telemetry streams. It contains computed values ready to be rendered or displayed to the end user.
- **Telemetry Reducers**² — This package contains the telemetry reducer extension point. A telemetry reducer aggregates low level software product and process data, and returns a collection of telemetry streams. To provide a custom implementation, a developer has to implement the “*TelemetryReducer*” interface and supply a declaration file in XML format. The Core_Telemetry module uses Java reflection to discover and load custom reducer implementations dynamically at runtime.
- **Telemetry Functions**³ — This package contains the telemetry function extension point. A telemetry function takes a telemetry stream collection as input, and returns another telemetry streams collection as output. To provide a custom implementation, a developer has to implement the “*TelemetryFunction*” interface and supply a declaration file in XML format. The Core_Telemetry module uses Java reflection to discover and load custom function implementations dynamically at runtime.

²Telemetry reducers are also known as telemetry reduction functions, because the grammar to invoke a telemetry reducer is the same as the grammar to invoke a telemetry function. However, the similarity is superficial since the underlying implementation is completely different.

³This definition excludes telemetry reduction functions.

- **Telemetry Evaluation Engine** — The telemetry evaluation engine walks the telemetry abstract syntax tree, resolves references to telemetry definitions, and invokes telemetry reducers and functions. The evaluation result is either a telemetry chart or a telemetry report.
- **Telemetry Definition Management Console** — This package implements a web interface for the telemetry definition management console, which allows a user to manage persistent definitions of telemetry streams, charts, and reports. The persistent definitions are used by the telemetry chart/report analysis, so that a user can select from a list of predefined charts or reports to perform the analysis.
- **Telemetry Analysis** — This package implements the web UI for the three telemetry analyses introduced in Section 4.2.1: the expert analysis, the report analysis, and the chart analysis.

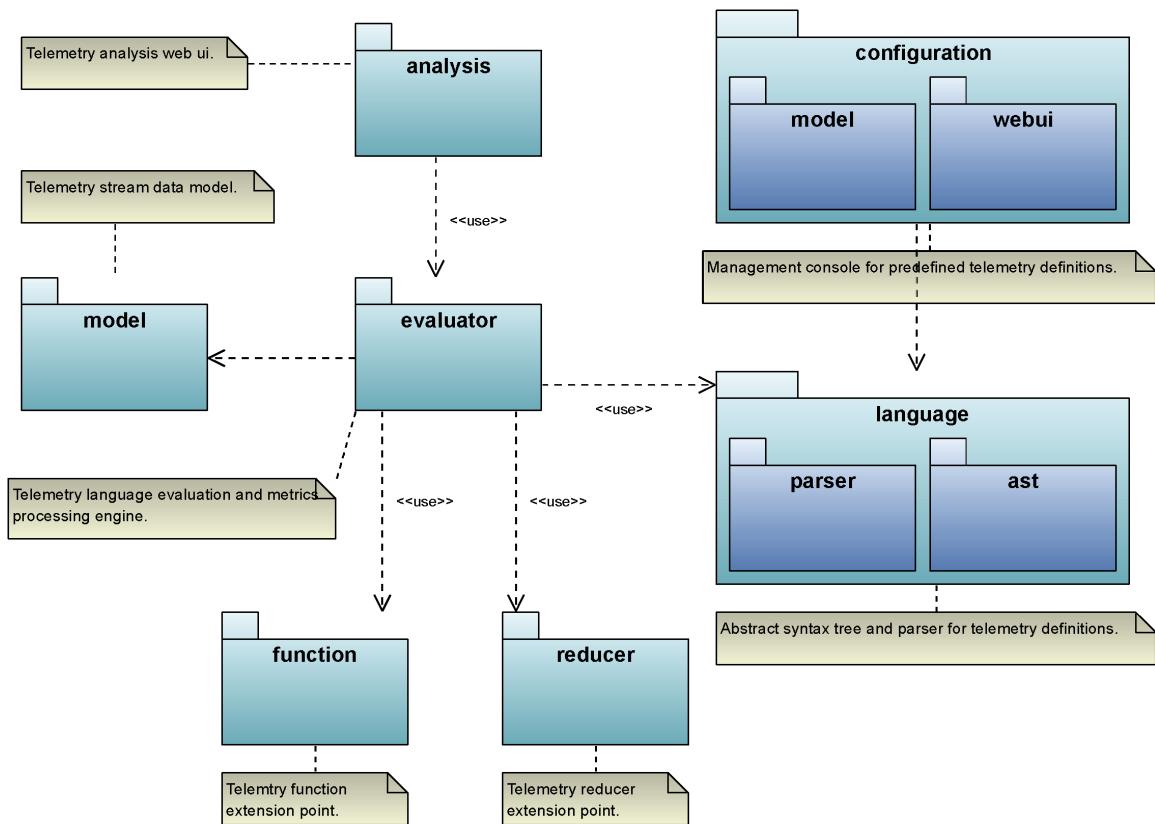


Figure 4.3. Core_Telemetry Module Package Structure

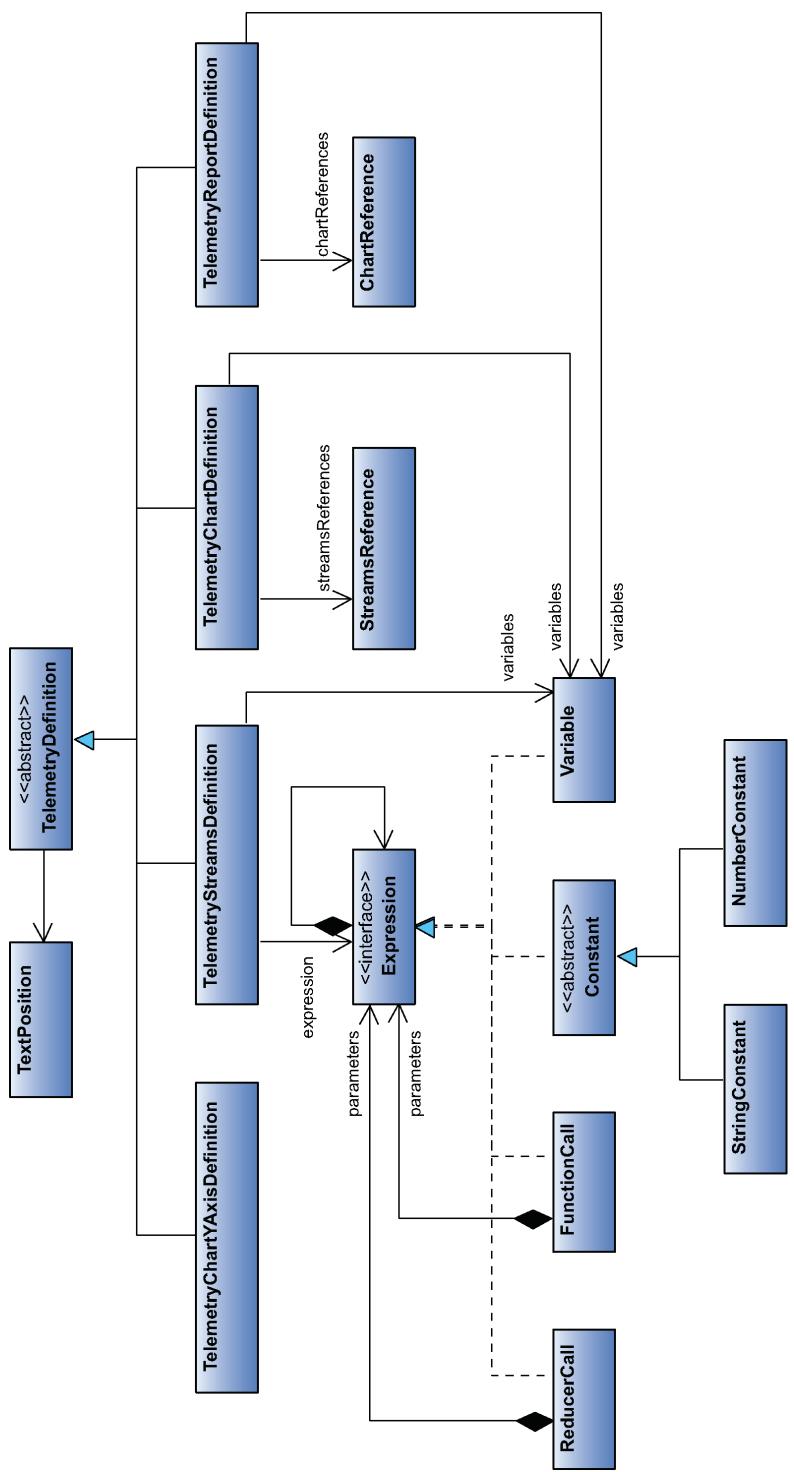


Figure 4.4. Telemetry Language Abstract Syntax Tree

4.3 Telemetry Reducers and Functions

The Core_Telemetry module is itself an extensible framework. It defines extension points for telemetry reducers and functions. The functionality of the telemetry module is determined by the available implementation of reducers and functions in a system. This section lists the reducers and functions that are currently available.

4.3.1 Telemetry Reducers

The following telemetry reducers are available:

- **ActiveTime Reducer:**

Computes single telemetry stream for project active time in hours.

- **Build Reducer:**

Computes single telemetry stream for build success count or build failure count.

- **CodeChurn Reducer:**

Computes single telemetry stream for code churn: lines added or lines deleted.

- **CodeIssue Reducer:**

Computes single telemetry stream for the number of potential issues in the code generated by tools like FindBugs [31] and PMD [69].

- **Commit Reducer:**

Computes single telemetry stream for code commit count.

- **CommitCycle Reducer:**

Computes single telemetry stream for the number of commits with specified local quality assurance behavior.

- **FileMetric Reducer:**

Computes single telemetry stream for source code size information.

- **IntegrationBuildFailure Reducer:**

Computes single telemetry stream for the number of integration build failures.

- **Issue Reducer:**

Computes single telemetry stream for the number of issues satisfying specified criteria.

- **JavaCoverage Reducer:**

Computes single telemetry stream for Java code unit test coverage.

- **JavaDependency Reducer:**

Computes single telemetry stream for Java code dependency.

- **LanguageFileMetric Reducer:**

Computes multiple telemetry streams for source code size information, one telemetry stream for each language found in the project.

- **MemberActiveTime Reducer:**

Computes multiple telemetry streams for member active time in hours, one telemetry stream for each member of the project.

- **MemberCodeChurn Reducer:**

Computes multiple telemetry streams for code churn, one telemetry stream for each member of the project.

- **MemberCommit Reducer:**

Computes multiple telemetry streams for code commit count, one telemetry stream for each member of the project.

- **MemberUnitTest Reducer:**

Computes multiple telemetry streams for the number of successful or failed unit test invocations, one telemetry stream for each member of the project.

- **Perf Reducer:**

Computes single or multiple telemetry streams for the values regarding a performance test. Multiplicity is determined by reducer parameter values.

- **ReviewActivity Reducer:**

Computes single or multiple telemetry streams for project review active time in hours. Multiplicity is determined by reducer parameter values.

- **ReviewFile Reducer:**

Computes single or multiple telemetry streams for the number of files reviewed. Multiplicity is determined by reducer parameter values.

- **ReviewIssue Reducer:**

Computes single or multiple telemetry streams for the number of issues uncovered through code reviews. Multiplicity is determined by reducer parameter values.

- **UnitTest Reducer:**

Computes single telemetry stream for the number of successful or failed unit test invocations in the project.

- **WorkspaceActiveTime Reducer:**

Computes multiple telemetry streams for active time in hours in top level workspaces, one telemetry stream for each top level workspace.

- **WorkspaceCodeChurn Reducer:**

Computes multiple telemetry streams for code churn in top level workspaces, one telemetry stream for each top level workspace.

- **WorkspaceCommit Reducer:**

Computes multiple telemetry streams for code commit count in top level workspaces, one telemetry stream for each top level workspace.

- **WorkspaceCoverage Reducer:**

Computes single telemetry stream for Java code unit test coverage in top level workspaces, one telemetry stream for each top level workspace.

- **WorkspaceFileMetric Reducer:**

Computes multiple telemetry streams for source code size information in top level workspaces, one telemetry stream for each top level workspace.

4.3.2 Telemetry Functions

The following telemetry functions are available:

- **Add Function:**

Internal stock function supporting telemetry language operator “+.”

- **Sub Function:**

Internal stock function supporting telemetry language operator “-.”

- **Mul Function:**

Internal stock function supporting telemetry language operator “*.”

- **Div Function:**

Internal stock function supporting telemetry language operator “/.”

- **Filter Function:**

User callable function that filters out telemetry streams in a telemetry stream collection according to specified ranking function and threshold value.

- **FilterZero Function:**

Usable callable function that filters out telemetry streams with values of zero or no value in a telemetry stream collection.

4.4 Chapter Summary

In this chapter, I have introduced my implementation of software project telemetry. It is implemented as a Hackystat extension module. More detailed information and the source code are available at the Hackystat public website: www.hackystat.org. The next chapter discusses the evaluation strategy for software project telemetry.

Chapter 5

Evaluation Strategy

There are a variety of possible approaches to empirical evaluation of software project telemetry. This chapter provides an overview of these approaches based on Creswell's book *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* [19]. The approach I chose is based on the concepts and techniques presented in that work. This chapter starts with a review of research methods in Section 5.1, followed by a discussion of the evaluation strategy with respect to software project telemetry in Section 5.2. Section 5.3 concludes the chapter.

5.1 Review of Research Methods

Creswell [19] categorizes research methods into three paradigms: *quantitative*, *qualitative*, and *mixed-methods*, according to their underlying philosophical assumptions about what constitutes knowledge and how knowledge is best acquired. The *quantitative* paradigm is related to *post-positivism*; the *qualitative* paradigm is related to *constructivism*; and the *mix-methods* paradigm is related to *pragmatism*.

5.1.1 Quantitative Paradigm and Post-Positivism

The philosophical underpinning of the quantitative paradigm is post-positivism. Post-positivism differs from positivism by recognizing that there is no absolute truth. Instead, it seeks to develop “relevant true statements” that can explain a situation and describe a causal relationship. 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.

The most often used inquiry strategy in the post-positivist paradigm is the experiment. There are three basic experimental designs. From the least rigid to the most rigid, they are: correlation study, quasi-experiment, and true experiment. In a correlation study, a single group is studied without comparison to an equivalent non-treatment group. The quasi-experiment design introduces a control group, but it falls short on random assignment of study subjects. The true experiment design employs both a control group and randomization. The purpose of increased rigidity is to control as many confounding variables as possible in order to determine true cause and effect. It is often thought that a true experiment is the only research method that can adequately measure the causal relationship. However, in the real world, a true experiment might be too expensive or might not be feasible at all.

The primary criteria with which to judge post-positivist research are *internal validity* and *external validity*. Internal validity is related to causality. It is demonstrated by showing that the cause not only correlates with but also precedes the effect, and that there is no plausible alternative explanation for the observed effect. External validity is related to generality. It is the degree to which the results obtained in a study can be applied to a larger population. Thus, for example, the use of control groups and randomization techniques are crucial to mitigate the threats to internal and external validity.

5.1.2 Qualitative Paradigm and Constructivism

The philosophical underpinning of the qualitative paradigm is constructivism. Constructivism assumes that all knowledge is “constructed” by observers who are the product of traditions, beliefs, and the social and political environment within which they operate. A researcher makes knowledge claims primarily based on constructivist perspectives, such as multiple meanings of individual experiences and socially or historically constructed meanings. The researcher tends to collect data through open-ended questions or by observing the participants’ behaviors, trying to understand a particular situation, event, role, group, or interaction from their views. 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 [64].

A major factor that distinguishes constructivism from post-positivism is that a researcher is not prescribing the questions that need to be answered from his / her own standpoint. Instead, the researcher tries to learn from the participants. In other words, the difference lies in the views about the nature of knowledge and how knowledge is best acquired. While hypotheses are specified *a priori* in the post-positivist paradigm, they are established *a posteriori* in the constructivist paradigm. Other unique characteristics of the constructivist paradigm are:

- A constructivist study usually takes place in natural settings where human behavior and events occur.
- A constructivist study often uses multiple interactive methods such as open-ended questions, observations, and interviews.
- A constructivist study focuses on the participants' perceptions, experiences, and their understanding of the world in which they live and work.
- A constructivist study is emergent rather than tightly pre-configured.
- A constructivist study places little importance on developing statistically valid samples, or on searching for statistical support for hypotheses.

A primary reason for conducting a constructivist 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. There are many established methods to conduct constructivist inquiries. For example, 28 methods were identified in [80], and 19 in [83]. Among them, the most commonly-used methods are observation, interview, case study, and grounded theory¹:

- In the observation method, a researcher gathers firsthand data on programs, processes, or behaviors being studied. The intent is to obtain a holistic picture of how people describe and structure their world in the context of the social settings they live in. There are various observation techniques. The most fundamental distinction is the extent to which a researcher is a participant in the setting being studied. It can range from complete involvement in the setting as a full participant, to complete separation from the setting as an outside spectator.

¹Some of them, such as case study and grounded theory, might be better called methodologies instead of methods. However, the difference in terminology is not important to this research.

- In the interview method, the assumption is that the participants' perspectives are meaningful, knowable, and able to be made explicit. There are two types of interview: structured vs. in-depth. A structured interview is essentially a carefully-worded questionnaire that follows a rigid form. The purpose is to ensure uniformity of interview administration. On the other hand, an in-depth interview encourages free responses for more detailed exploration of open-ended questions.
- In the case study method, a researcher explores one or more cases (a program, an event, an activity, a process, etc.) in depth in order to gain a sharpened understanding of why the instance happened as it did, and what might become important to look at more extensively in future research. It lends itself especially to generating rather than testing hypotheses. The emphasis of a case study is on investigating a few cases in detail, rather than using large samples and following a rigid protocol to examine a limited number of variables.
- In the grounded theory method, a researcher generates theories from data. The goal is to formulate hypotheses based on conceptual ideas from empirical data. The basic approach is to read and re-read a textual database such as a collection of field notes in order to label variables and note their interrelationships. Formally, the approach includes steps of coding (open coding, selective coding) and memoing (theoretical memoing). When the method is followed correctly, the researcher should be able to generate a theory that fits the data perfectly.

The distinctions between these methods are blurred at best. They are not mutually exclusive. They just have different emphases. For example, the observation and interview methods focus more on data collection, while the grounded theory method focuses more on hypothesis generation. Therefore, it is possible to say that *"I am conducting a case study, collecting data through observation and interview, and generating hypotheses following the grounded theory method."*

The goal of constructivist research is quite different than that of post-positivist research. Instead of controlling the context by using random assignment and control groups in order to develop "truth" that is "broadly" applicable outside the context in which the research occurred, constructivist research focuses on "deep" understanding of the context in which the observed events and outcomes occurred. The results in a constructivist study are thus closely tied to the context in which the research was carried out.

As a result, the criteria with which to judge constructivist research are quite different than that for post-positivist research. Internal validity carries a different meaning than that in a post-positivist research. Since the results in a constructivist study are actually an integrated set of conceptual hypotheses emerging from empirical data in the particular context the study was conducted, internal validity in its traditional sense is consequently not an issue [33, 34]. In other word, it no longer focuses on causality. Instead, a constructivist study should be judged by the degree to which its results fit the existing data. External validity also carries a different meaning. Since the results in a constructivist study are somewhat tied to the context in which the research is conducted, they may not be applicable in another context. While in the post-positivism paradigm, external validity focuses on the extent to which the results obtained in one study can be generalized to a larger population; in the constructivism paradigm, it focuses more on whether the process used in acquiring the knowledge could work in a different setting. Therefore, most constructivist studies seek to provide *thick descriptions*,² so that anyone who is interested in transferability of the results can have a solid framework for comparison [63].

5.1.3 Mixed-Methods Paradigm and Pragmatism

Once the relationship between the quantitative paradigm and the qualitative paradigm is clear, the mixed-methods paradigm is easy to comprehend. Strictly speaking, it is not really a paradigm of its own. It is just a mix of different methods in the same research. The methods in the mix can come from either the post-positivist tradition, or the constructivist tradition, or both.

The philosophical underpinning of the mixed-methods paradigm is pragmatism, in which knowledge claims arise out of actions, situations, and consequences rather than antecedent conditions such as those in post-positivism. To mixed methods researchers, understanding the problem and finding the solutions are more important than commitment to a single methodology. As a result, they use whatever methods that are available in order to best understand the problem and find solutions.

According to Creswell [19], 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 “multi-method matrix” to examine multiple approaches to data collection in a study, which prompted other researchers to start mixing methods. Soon approaches

²According to *wikipedia*, thick description is a phrase used most famously by the anthropologist Clifford Geertz to describe his own work: explaining the context of the practices and discourse that take place within a society such that these practices become meaningful to an “outsider.”

associated with field methods such as observations and interviews were combined with traditional surveys.

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

- A mixed-methods researcher does not mix different methods blindly. There is always a purpose for “mixing.”
- A mixed-methods researcher is “free” to choose the methods, techniques, and procedures of research that best meet his/her needs and purposes, rather than subscribing to only one way.
- A mixed-methods researcher uses different methods because they work to provide the best understanding of the research problem.

5.1.4 Clarification of Terminologies

Before discussing my approach to the evaluation of software project telemetry, I will first try to clear some terminology confusions around “quantitative vs. qualitative.”

To reiterate briefly, *post-positivism* is the philosophical underpinning of the *quantitative* paradigm. It seeks to develop “relevant truth” that can explain a situation and describe a causal relationship. *Constructivism* is the philosophical underpinning of the *qualitative* paradigm. It assumes that all knowledge is “constructed” by observers who are the product of traditions, beliefs, and the social and political environment within which they operate.

Creswell did a good job of distinguishing the various kinds of research methods according to their underlying philosophy about knowledge. However, many researchers, including Creswell himself, overloaded the phrases like “quantitative research” and “qualitative research” with multiple meanings. Sometimes they were used to refer to the post-positivist and constructivist paradigms to acquire knowledge, while other times they were used to refer to the collection and analysis of quantitative (numeric) and qualitative (non-numeric) data. The wide-spread use of the terminologies like “quantitative research” and “qualitative research” creates confusion, because either paradigm for acquiring knowledge can use either numeric or non-numeric forms of data. The only relationship, at best, is the historical tendency that most post-positivist research focused on collection and

analysis of quantitative data, while most constructivist research collected and analyzed qualitative data. However, this is not a rule at all. There is nothing to prevent a constructivist study from using quantitative data, or vice versa. What exacerbates the problem is that, in most cases, quantitative data and qualitative data are convertible to each other, though the conversion process might result in loss of information. The distinction between quantitative and qualitative is thus almost meaningless. As a result, the title of Creswell's book [19] would be much more self-evident and accurate if it would have been called "*Research Design: Post-positivism, Constructivism, and Mixed Methods Approaches*" instead of "*Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*."

For the sake of clarity, I will avoid the use of "qualitative" and "quantitative" when discussing my research, and instead use more precise terms for my methods (e.g., grounded theory) and data (e.g., questionnaire).

5.2 Software Project Telemetry Evaluation Design

Generally speaking, the post-positivist paradigm is more suitable for natural science inquiry, while the constructivist paradigm is more suitable for social science exploration. Software engineering is neither a pure natural science nor a pure social science. It lies somewhere on a continuum between the two. Therefore, in setting up the evaluation of software project telemetry, I considered both options:

- If I followed the post-positivist paradigm, my evaluation would start with some hypotheses about software project telemetry, such as how it might affect the project decision-making process. An experiment would then be set up to collect the relevant data, and statistical procedures would be used to analyze them. I would try to control the software development environment in which the experiment was conducted through the use of control groups and random assignments. My goal would be to test a theory about the use of software project telemetry that would be broadly applicable in most software development environments, if not all of them.
- If I followed the constructivist paradigm, my evaluation would start with open-ended data collection. Knowledge about software project telemetry would be acquired *a posteriori* after

the data collection was complete. I would emphasize the importance of understanding the software development environment in which software project telemetry was adopted, rather than trying to control the environment through control groups and random assignments. My goal would be to gain a much deeper understanding about the use of the technology within the particular environment in which the study would be carried out, and perhaps even generate a theory from the data that would explain what I observed in that context.

The two paradigms provide a trade-off between breadth and depth with respect to the knowledge to be acquired about the use of software project telemetry. The post-positivist paradigm aims to yield broadly generalizable conclusions. But, at very best, it can elicit only a few and probably superficial guidance about how to use the technology effectively. On the other hand, the constructivist paradigm aims to include many more detailed insights. But such insights might be limited in the extent to which they apply beyond the specific software development environment in which they are generated.

It is this trade-off that determined my strategy for the evaluation of software project telemetry. The post-positivist paradigm requires at least some basic level of experience with the technology, so that meaningful hypotheses could be specified *a priori*. In case where there is little experience, the constructivist paradigm would be more appropriate. It would allow me to gain a sharpened understanding within the particular software development environment in which the technology is deployed, so that hypotheses could be generated from the events occurred in that environment. This could, in turn, provide me with valuable clues in deciding what might be important to study more extensively in the future.

Software project telemetry is a brand-new approach to metrics-based project management and process improvement. Up to now, only its theoretical properties are clear. They are the principles upon which the approach is developed (see Chapter 3). To reiterate, software project telemetry is designed to address the “*metrics collection cost problem*” through highly automated measurement machinery: software sensors are written to collect metrics automatically and unobtrusively. Sensors keep metrics collection cost low by eliminating the chronic “context-switch” overhead. Software project telemetry is also designed to address the “*metrics decision-making problem*” through a domain-specific language for the representation of telemetry trends for different aspects of software development. Project management and process improvement decisions are made by detecting changes in telemetry trends and comparing trends in two different periods of the same project,

which not only eliminates the need to build statistical models that require frequent calibration but also enables empirically-guided in-process control of a project that is still being developed.

The theoretical properties of software project telemetry appear promising as it is described on paper, since it overcomes many limitations in existing metrics-based approaches to project management and process improvement. But what will happen when it is used in “real world” settings? Software project decision-making is a very complicated phenomenon. It involves not only the software product being developed and the process being followed, but also human behaviors and interactions among the developers. At this early stage of research, there is little empirical experience with respect to the real world use of software project telemetry. Many questions remain to be answered. For example, will software project telemetry actually be useful to a software development team in practice? What impact will it have on project decision-making? What are the best practices? What are the obstacles? Which obstacle can be fixed, and which might become technology adoption barrier?

While it would be possible to set up a controlled experiment in the post-positivist paradigm to compare the decision-making value of software project telemetry to those of other competing metrics-based approaches, such as PSP, TSP and CMM, I feel it is not the most useful form of research at this point, because I am not even sure whether a software organization will want to use the technology if it is given the opportunity. Instead, what would be most fruitful at this stage is to conduct research in the constructivist paradigm, which has techniques for generating the appropriate kind of understanding of software project telemetry in and of itself. By exploring how the technology is used in some real software development environments, I can gather detailed information about how it helps developers and managers make decisions and learn from these experiences. A comparative study is best performed after this initial understanding has been acquired. For example, if it turns out that software project telemetry is indeed adoptable and useful in the environment in which it is tried, then the experience from that study could be used to guide the design of further evaluation of software project telemetry, which might be a comparative study in the post-positivist paradigm.

The major disadvantage of following the constructivist paradigm in evaluating software project telemetry is the generalizability of the results. Since a constructivist study will focus on the particular software development environment in which the technology is adopted, the knowledge acquired in that environment may not be applicable in other environments. The reality is that software development environments are diverse, and each may have different development processes and

constraints on metrics collection and analysis. For example, in COCOMO II, the post-architecture model has 17 effort multipliers and 5 scale factors to approximate this diversity. Achieving some degree of generalizability is important to wide adoption of software project telemetry in the future. An approach to mitigate this disadvantage is to conduct evaluations in different environments with complementary characteristics. The idea is that by comparing and contrasting the similarities and differences of the experiences from different environments, one can “interpolate” and “extrapolate” the results to gain further insights that might generalize to other software development environments.

My final choice is to follow the mixed-methods paradigm. The use of multiple methods has the advantage of using the strength in one method to make up for the weakness in another method. The primary goal of my evaluation is to assess metrics collection cost and decision-making value of software project telemetry. The secondary goal is to discover obstacles the developers might encounter during their use of the technology, and to gain insights about software project telemetry best practices and possible technology adoption barriers. The two software development environments are:

- **Classroom** — This is the two software engineering classes taught by Dr. Philip Johnson at the University of Hawaii in Spring 2005: one class for senior-level undergraduate students, and the other for introductory-level graduate students. By curriculum design, the students were divided into groups of 2 - 4 members collaborating on group projects and introduced to use software project telemetry to collect metrics and perform analyses on their own data. There were 25 students participating the study: 9 from the undergraduate session, and 16 from the graduate session. The details of the classroom setting will be described in Section 6.1.
- **CSDL** — This is the Collaborative Software Development Lab at the University of Hawaii. It is a software engineering research lab. The study was conducted in Spring 2006 when a large scale software system (i.e., the Hackystat itself) with almost 300,000 lines of code was being developed and maintained by a team of five on-site developers and a project manager. Three of the developers were Ph.D. students (including me) in software engineering. They were hired by the lab working 20 hours a week. The other two were undergraduate students in their final semester. They were top students from the undergraduate software engineering class. They were working for the lab in exchange for personal development and course credit. The details of the CSDL setting will be described in Section 7.1.

The software development environments in the classroom and CSDL were quite different. In the classroom, there were a relatively large number of participants (25 students). They were working on small scale class projects. In CSDL, there were a relatively small number of participants (five developers and one project manager). However, the project under development was much larger in scale. It contained almost 300,000 lines of code in total, and had been under development for five years. The CSDL developers had significantly more software engineering experience and process maturity compared to the average student in the classroom.

As a result, the way that software project telemetry was introduced in the two environments was different. The classroom study was “*passive*” in nature: though the students were asked to use software project telemetry to collect metrics and perform analyses on their own data, I did not make any deliberate attempts to help them improve their software development processes. On the other hand, the CSDL study was “*active*” in nature: I introduced software project telemetry as a metrics-based process improvement program; I helped the project manager institute changes to improve project management practices; I also helped the developers gain insights into their development process.

Consequently, different data collection and analysis techniques were used in the two studies. The classroom study was relatively simple. My goal was to gather insights from a relatively large number of developers in a relatively short period of time. I distributed a questionnaire at the end of the semester to collect the student’s opinions about software project telemetry. To increase my confidence in the validity of their self-reported opinions, I also analyzed their telemetry analysis invocation pattern to determine the extent to which their opinions were based on the actual system usage. In the CSDL study, I pursued a much more in-depth data collection and analysis strategy over a much longer period of time. I collected data from observations and interviews. I generated hypotheses from the data. I also tested the hypotheses in a limited way by making changes to the telemetry system or implementing new facilities to see whether the hypothesized outcome would come true or not.

5.3 Chapter Summary

In this chapter, I have provided a review of research methods. The most important distinguishing factor among different methods is their underlying philosophy about knowledge. Post-positivism

and constructivism have very different view on the nature of knowledge and thus very different approaches to acquire knowledge. My evaluation of software project telemetry was carried out in two case studies following the mixed-methods approach. The next two chapters reports on the details of the two studies.

Chapter 6

Classroom Study

This chapter reports on an empirical study of software project telemetry in a classroom setting. The study was conducted in the two software engineering classes taught by Dr. Philip Johnson at the University of Hawaii in Spring 2005: one class for senior-level undergraduate students, and the other for introductory-level graduate students. By curriculum design, the students were divided into groups working on group projects, and introduced software project telemetry as a technique for collecting metrics and performing analyses on their own data. There were 25 study participants. At the end of the study, I distributed a questionnaire to collect the students' opinion about software project telemetry. I also analyzed their telemetry system usage pattern to cross-validate the extent to which their opinions were based on the actual system usage.

This chapter begins with a description of the classroom setting in Section 6.1. Section 6.2 describes my role in the study. Section 6.3 elaborates on the study design. Section 6.4 describes data collection and analysis procedures. Section 6.5 reports the results. Section 6.6 concludes the chapter with a summary of the insights learned from this study.

6.1 Classroom Setting

The study was conducted in two software engineering classes: one at the senior undergraduate level (ICS 414), and the other at the introductory graduate level (ICS 613). The two classes followed the same basic curriculum, except that the students at the graduate level were expected to read more supplementary materials and do a more thorough job with their programming assignments. The curriculum had two equally important components:

- **Software Lifecycle Techniques** — The curriculum covered software application lifecycle techniques. They included requirement management, design patterns, change and configuration management, code review and testing. The students were divided into teams of two to four members working on different projects, using tools such as Eclipse (a Java IDE), Ant (a Java build tool), CVS (a configuration management system), and JUnit (a Java unit test framework). The focus was on agile development practice.
- **Software Process Improvement** — The students were required to collect and analyze their software process and product metrics while performing development tasks. The purpose was to help them acquire hands-on experience in collecting, analyzing, and interpreting software metrics in order to improve their software development processes.

The students' software product and process metrics were collected and analyzed using the implementation of software project telemetry introduced in Chapter 4. This was the second semester that the system was used in software engineering classes. A pilot study took place in Fall 2004.

The software project telemetry system was deployed on a university server where all students created an account to access their metrics data so that they could run telemetry analyses to gain insights into their software development processes. In order to gather product and process metrics, the students were required to instrument their development tools with sensors. These sensors collected a wide variety of information such as the time each project member spent editing source code, the size metrics, the occurrence of unit tests, and the resulting test coverage.

The software project telemetry implementation provides three analysis interfaces: *telemetry chart analysis*, *telemetry report analysis*, and *telemetry expert analysis* (See Section 4.2.1). The telemetry chart and report analyses use predefined telemetry definitions, while the telemetry expert analysis requires the user to use the telemetry language to interact with the system directly. The instructor and I felt that introducing the telemetry language would impose an unnecessary burden on the students. Therefore, we predefined a set of charts and reports that we thought were most useful to them, and only the telemetry chart and report analyses were introduced in class.

6.2 Researcher's Role

The instructor of the two software engineering classes in which this 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 classes.

6.3 Study Design

The software engineering class was an environment where the use of software project telemetry was mandatory by curriculum design. An important goal of the class was to let the students gain experience with metrics-driven process improvement. This provided a good opportunity for me to gather opinions about software project telemetry from a relatively large number of users with relatively diverse backgrounds.

The study was a mixed-methods study. The goal was to explore the way the students used software project telemetry and the problems they encountered during their use in a natural setting. There were 25 students enrolled in the two classes. Following them around and observing how they interacted with the technology to make process improvement decisions was not a feasible choice. Therefore, I decided to use a survey to collect their opinions. The interview method was not chosen to conduct the survey because of the concern that the students might feel pressured to give more favorable opinions due to the involvement of the professor in the research, and thus bias the study results. Instead, a questionnaire was used. It has the advantage of ease of administration and rapid turn around in data collection. The disadvantage is that the questions I can ask are fixed and limited in a questionnaire.

The questionnaire was administered less than one week before the final examination. To further mitigate the threat that the students might be concerned that their responses would influence their grades and thus comment more favorably toward software project telemetry, I made the questionnaire anonymous, instructed the students specifically not to reveal their names in their responses, and assured them that their responses would only be processed after their instructor had turned in their final grades.

The decision to use a questionnaire, at the same time, implied that I had to rely on the students' self-reported opinions to evaluate software project telemetry. The threat was mitigated through cross-validation. The usage of the software project telemetry system was logged. If the survey responses were mostly positive and the log indicated that the students ran the analyses frequently or regularly, or if the survey responses were mostly negative and the log indicated that the students stopped using the system after a while, I could put more confidence in the responses because their opinions were consistent with the actual system usage pattern. On the other hand, if it turned out that the students' opinions were very positive but they all stopped using the system after a while, then I had to question their responses because the actual system usage pattern was inconsistent with their opinions. My cross-validation was partial in nature, because I could not match the students' telemetry analysis invocation data to individual responses due to the chosen design of anonymous questionnaire.

6.4 Data Collection and Analysis

This study collected data from two sources:

- A survey questionnaire was distributed on the last day of instruction asking the students their opinions about software project telemetry.
- The software project telemetry system was instrumented. All usage information was logged.

The two sources of data were integrated at data interpretation phase, and priority was given to the analysis of the questionnaire responses. The software project telemetry system log was used to assess the extent to which the students' opinions were based on the actual system usage.

6.4.1 Survey Questionnaire

The survey was conducted through an anonymous written questionnaire administered on the last day of instruction. The questions covered metrics collection overhead, analysis usability and utility, and the students' perception of 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*”. Then, I asked the students to rank their feelings toward the statement:

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree
- Not Applicable

The last option “*not applicable*” was provided to allow the students to skip the questions which they were unable to answer or they did not want to answer. At the end of each question, I provided large empty space for them to write down any related comments such as justification or elaboration of the answer. The last part of the questionnaire was a free response section where the students were encouraged to provide any additional suggestions or comments. The actual questionnaire is attached in Appendix B for reference.

6.4.2 System Usage Log

The software project telemetry system exposes a web interface through which the users could invoke telemetry analyses over their software product and process metrics. The system deployed for classroom use was instrumented with an automatic logging facility. Telemetry analysis invocation information, such as time, user name, and full web request parameters, were logged.

6.5 Results

All of the 25 students enrolled in the two software engineering classes participated in the questionnaire, of which 9 were from the senior undergraduate level class and 16 were from the introductory graduate level class. The students had a fairly diverse background. Their total programming experience, as defined from their first “Hello World” application, ranged from 3 to 25 years, with a mean of 6.92 and a standard deviation of 4.43. Their paid professional experience¹ ranged from 0 to 8 years, with a mean of 1.27 and a standard deviation of 2.10.² The survey was conducted in a normal class session on the last day of instruction, and the response rate was 100%.

6.5.1 Results from Individual Survey Question

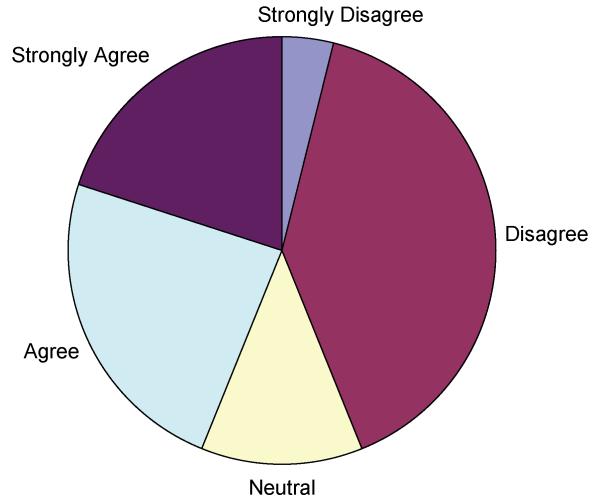
The survey questions are listed below along with the results. Each question was in the form of a statement, and the students were asked to choose the answer that most closely matched their feelings about the statement. Their choices were: *strongly disagree, disagree, neutral, agree, strongly agree*, and *not applicable*. The resulting statistics were computed by excluding the “*not applicable*” answers.

¹I specifically asked the students to exclude the experience from half-time or less than half-time on-campus employment, such as student helper or research assistant, even if they were paid to program.

²One student did not answer this single question and thus was not included in the statistics regarding professional background.

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

Elaboration of the Statement: Software project telemetry uses sensors to collect metrics. The sensors must be installed and properly configured by the developers. The question was designed to gather information about the one-time installation and setup cost of the sensors.



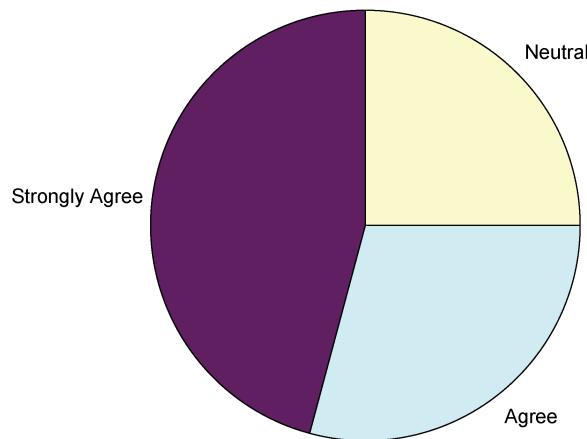
Response Rate: 25 out of 25

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

Explanation of the Result: It turned out that sensor installation and configuration involved quite complex procedures. 40% of the respondents did not agree with the statement. One of the students even responded: "*I still have problems with some 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.

Elaboration of the Statement: Once installed, sensors collect metrics automatically. This question was designed to gather information about the long term chronic metrics collection overhead (excluding the one-time setup cost).



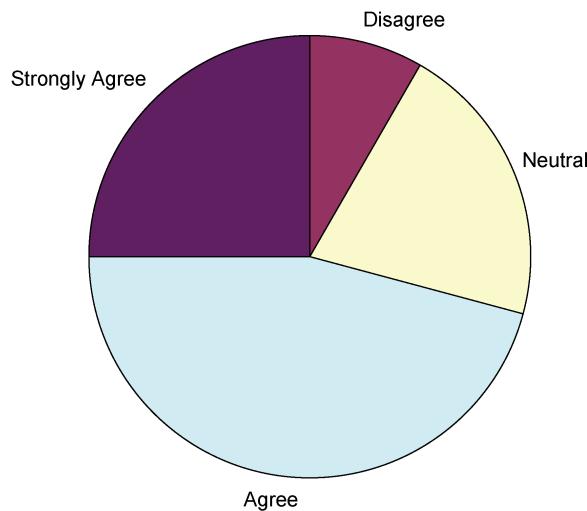
Response Rate: 24 out of 25

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

Explanation of the Result: 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.

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

Elaboration of the Statement: The telemetry chart and report analysis used in the class did not require the students to use the telemetry language. Instead, the instructor and I predefined a set of useful telemetry charts and reports for them. This question was designed to gather information about the usability of the two telemetry analysis interfaces.



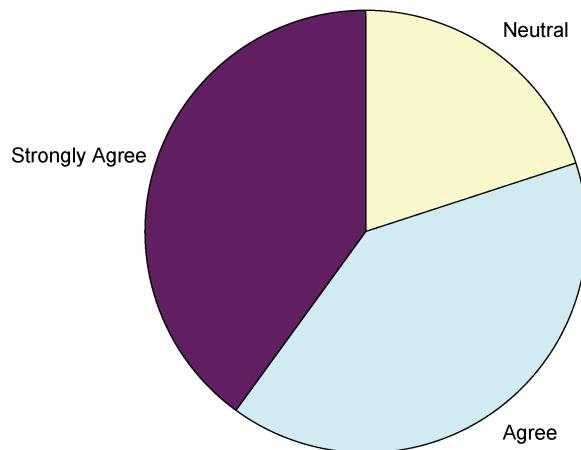
Response Rate: 24 our of 25

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

Explanation of the Result: Though most students appreciated the idea of being able to run metrics analysis by choosing from a list of predefined telemetry charts and reports, they thought the telemetry analysis interface could be further improved. It turned out that a major problem involved the input of analysis parameter values. Different telemetry charts or reports had different parameter requirements, but it was hard to tell from the simple web interface what parameters were expected.

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

Elaboration of the Statement: One of the goals of software project telemetry is to make development process transparent so that problem can be detected early. This question was designed to measure whether software project telemetry had achieved that goal or not.



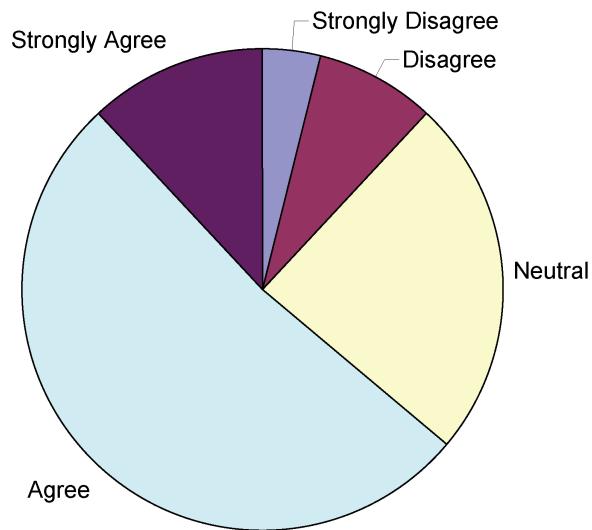
Response Rate: 25 out of 25

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

Explanation of the Result: No one disagreed with the statement. The numbers appeared to indicate that software project telemetry had achieved the goal of making development process transparent. In fact, some of the students appeared to suggest that it made their process more transparent than they had wished by expressing concerns about their data privacy.

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

Elaboration of the Statement: This question was designed to ask the students whether there was any self-perceived process improvement as a result of using software project telemetry. In other words, it tried to determine the causal relationship between software project telemetry and process improvement.



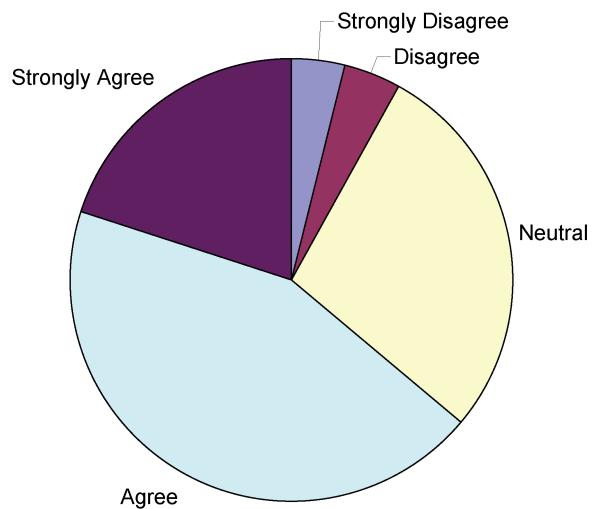
Response Rate: 25 out of 25

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

Explanation of the Result: Most students thought that telemetry analyses had helped them improve their software development process. However, due to the limitation of this study, there is no conclusive evidence to determine whether the students' self-perceived process improvement was due to the use of telemetry analyses on their metrics, 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.

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



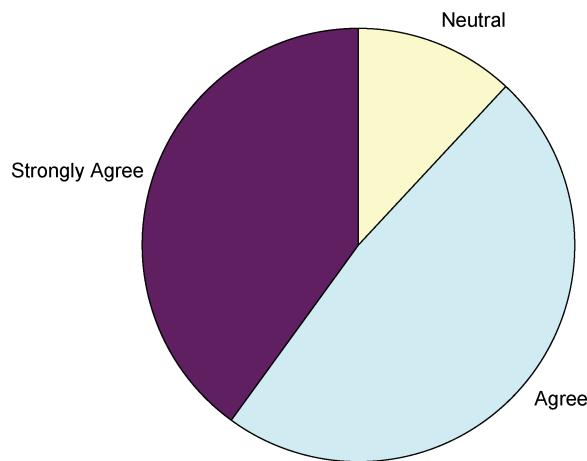
Response Rate: 25 out of 25

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

Explanation of the Result: The majority of the students confirmed the value of software project telemetry as an approach to metrics-based 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.*”

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

Elaboration of the Statement: This question was designed to ask the students whether they perceived software project telemetry as a reasonable approach to project management and process improvement in “real” development settings from the perspective of *a project manager*.



Response Rate: 25 out of 25

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

Explanation of the Result: The data appeared to overwhelmingly suggest that software project telemetry was a valuable tool for project managers. The result seemed to be consistent with the finding in Statement 4 that software project telemetry achieved the goal of making development process transparent. It also seemed to confirm the recurring theme in the answers to Statement 6 that developers worried about their privacy.

6.5.2 Results from Free Response Section

The students provided a lot of textual feedback. I identified several themes: some were positive, some were negative. They are all listed below in bold face followed by the actual comments from the students.

Metrics Collection:

- **Sensor-based automated and unobtrusive metrics collection is effortless.**

“Overall it is an incredible tool that generally makes software development metric collection effortless.”

- **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 to work correctly.**

Comment: This is a complex issue. Several reasons have been identified 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.”

Metrics Analysis:

- **Software project telemetry makes development process transparent.**

“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?”

- **Telemetry data interpretation is the key to get the value from the tool.**

“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.”

- **Software project telemetry is better suited as a manager’s tool than as a developer’s tool.**

“(It is) more useful for management.”

“I want to know what people are doing.”

- **The web interface for telemetry analysis can be made 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 parameters needed were confusing.”

Privacy Concern:

- Developers have concerns about the privacy of 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.”

Uncategorized:

- Uncategorized

“Eclipse sensor updates too often. Every 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.”

6.5.3 Results from Telemetry Analysis Invocation Log

The automatic logging facility in the software project telemetry system recorded all analysis requests in space-delimited files. I wrote a simple application parsing the log files and imported the data into a Microsoft Access database. I then ran SQL query inside Access and exported the results into a Microsoft Excel file. From there, I used pivot table slicing the data. The final results were presented in Figure 6.1 and 6.2.

Figure 6.1 showed the total number of telemetry analysis invocations by all the students participating in this study. The number of invocations increased steadily from 47 in January to 967 in April. However, there was a sharp decrease in the number in May, which was 101. The decrease could be explained by the fact that Spring semester ended in that month and final examinations were all scheduled in the second week by the University.

The numbers clearly showed a pattern of increased utilization of telemetry analysis over time by the students. There were two plausible explanations: (1) telemetry analysis got more interesting when there were more metrics data available, and (2) it took some time for the students to realize the benefit of software project telemetry.

Figure 6.2 broke the numbers in Figure 6.1 further by individual student. Overall, the individual student's telemetry analysis invocation followed the same pattern as in Figure 6.1: steady increase in the number from January to April, and a sharp drop in May due to final examinations. There were 25 study participants, but Figure 6.2 indicated that one student had never invoked telemetry analysis. Since the survey questionnaire was anonymous, I was unable to match the individual student's analysis invocation information to his/her survey response.

Interestingly, 60% of the students did not invoke telemetry analysis during the first two months: January and February. However, in March and April, almost everybody invoked telemetry analysis a lot (of course, except the 25th student who had never run a single analysis). The numbers seemed to suggest that software project telemetry was indeed useful to the students once their historical metrics data had reached certain threshold. It was no surprise, since historical metrics data were used to establish a baseline for comparison in software project telemetry.

My final conclusion was that the students' telemetry analysis invocation pattern were consistent with the overall positive tune in their survey responses. It assured my confidence in the subjective opinions from the students about software project telemetry.

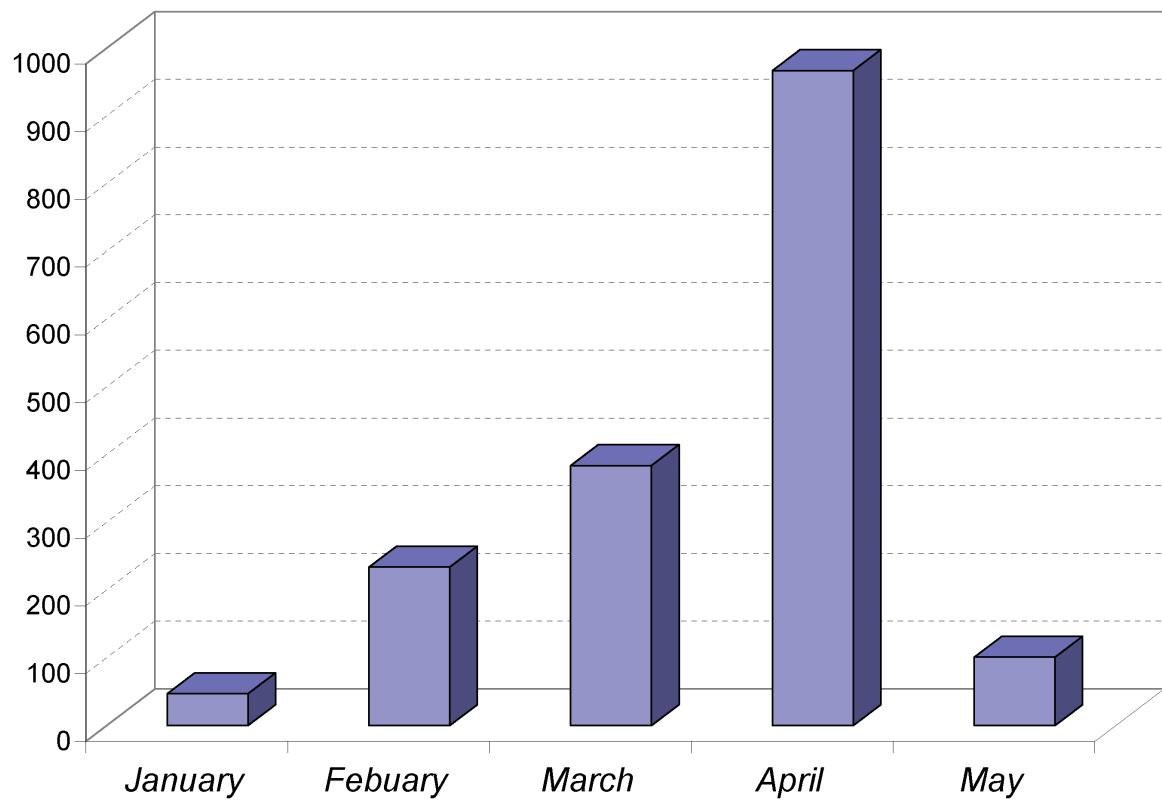


Figure 6.1. Telemetry Analysis Invocation by Month

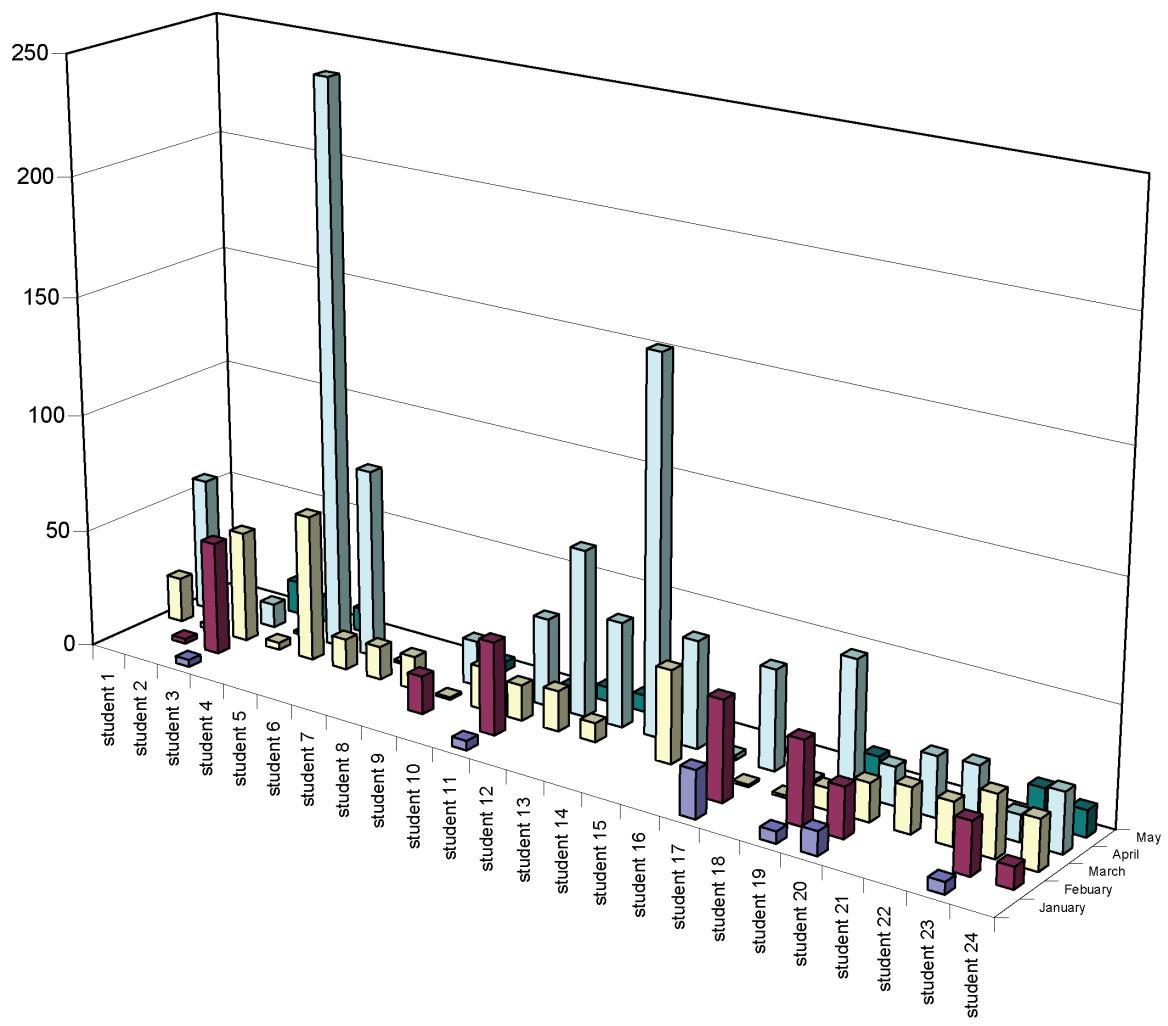


Figure 6.2. Telemetry Analysis Invocation by Individual and Month

6.6 Study Conclusion

This study yielded a number of valuable insights into software project telemetry and its implementation.

An automated and unobtrusive metrics collection mechanism is crucial to the success of a metrics program. From the student's feedback, the sensor-based approach appeared to have achieved the goal of eliminating long-term chronic overhead related to metrics collection. However, the one-time setup cost of the sensors was still too high. Though it was not a major issue in the classroom setting, it could cause significant adoption barrier in other environments. Many students had expressed the wish to have an all-in-one intelligent graphical user interface to install and configure the sensors easily. Fortunately, such an installer is now available as a result of user feedback.

Software project telemetry appeared to have achieved its design goal of making development process transparent. Most students agreed that they were made more aware of both their own and their team's development process as a result of using the system. However, there was no conclusive evidence to determine whether the increased awareness had actually helped the students improve their software development processes. The data appears to indicate that the ability to understand and interpret telemetry data is related to whether software project telemetry is useful. There were several incidents in which the sensors did not seem to collect metrics correctly, or the telemetry analyses did not seem to compute the data as expected. Some of these were caused by inappropriate interpretation of the results. It seemed that effort-related metrics were most susceptible to misinterpretation. As far as the implementation was concerned, many students suggested that the web interface for telemetry analysis worked but could be made more user-friendly.

There was a data privacy issue. Some students seemed to suggest that software project made their development process more transparent than they had wished. They concerned that their personal process data might be misused. I was very well aware of the issue when implementing the system, and had taken steps to limit the kinds of data that could be accessed by people other than the person owning the data. However, it seemed hard to reconcile the conflicting requirements between project management and privacy protection. Some students expressed that they would not want to share personal metrics with others, while other students said they would like to know what other people were doing.

Chapter 7

CSDL Study

The previous chapter reported on a classroom study of software project telemetry. The classroom study has the advantage of obtaining insights from a relatively large number of people in a relatively short period of time. However, it has a number of limitations: the size and the scope of the class projects were relatively small, the time the students could devote to software development activities was relatively limited, and the students in classroom tend to have less software engineering experience. To mitigate the limitations, I performed another study in the Collaborative Software Development Lab (CSDL) during Spring 2006. The CSDL study had a number of complementary characteristics to the classroom study. Though it involved only five developers and one project manager, the system under development was much larger in scale with almost 300,000 lines of code in total. It had been under development for five years. The CSDL developers had significantly more software engineering experience compared to the developers in the classroom on average. Instead of a one-shot survey, I pursued a much more in-depth data collection and analysis strategy over a much longer period of time. While still within an academic setting, it intends to provide data that reflect an environment much closer to those in industrial settings.

This chapter begins with a description of the CSDL setting in Section 7.1. Section 7.2 describes my role in the study. Section 7.3 elaborates on the study design. Section 7.4 describes data collection and analysis procedures. Section 7.5 reports the results. Section 7.6 concludes the chapter with a summary of the insights learned from the study.

7.1 CSDL Setting

The Collaborative Software Development Laboratory (CSDL) is a software engineering development and research lab at the University of Hawaii. Its mission statement, as published on its website, is “to provide a physical, organizational, technological, and intellectual environment conducive to collaborative development of software engineering skills.”

CSDL has been focusing on the development of Hackystat since 2001. Hackystat is a framework for automated metric collection and analysis of empirical software engineering product and process metrics. Several extensions have been developed based on this framework, and are being maintained by the lab. Some of the extensions are specialized to high-level software process analysis, such as Software Project Telemetry as the result of this thesis research, and CGQM for continuous machine-executable Goal-Quality-Metric paradigm. Other extensions are specialized to low-level software process analysis, such as SDSA for micro-process views of software development behaviors at the time scale of minutes or hours, and HPCS for bottleneck identification in the development of parallel programs for high performance computers. Thus, the scope of the Hackystat project and its framework, includes, but is also much broader than, Software Project Telemetry.

At the time this study was conducted, the Hackystat framework and the extensions maintained by CSDL constituted nearly 300,000 lines of code in total. Figures 7.1 shows the breakdown of size by programming languages. The code was organized into over 70 different modules. The development team consisted of one project manager and five on-site developers. The project manager was Dr. Philip Johnson. He was the lab director controlling the overall direction of Hackystat. He also spent a considerable amount of time working on code himself. Three of the developers were Ph.D. students (including me) in software engineering. They were hired by the lab working 20 hours a week. The other two were undergraduate students in their final semester. They were the top students from the undergraduate software engineering class. They were working for the lab in exchange for personal development and course credit.

The team adopted agile software development methods, emphasizing working software as the primary measure of progress. The project sources were stored in a *Subversion* repository,¹ supporting concurrent development by allowing multiple developers to checkout the sources and commit their changes at the same time. The developers tended to work with a subset of the source code rel-

¹*Subversion* is a version control system for the management of multiple revisions of the same unit of digital information.

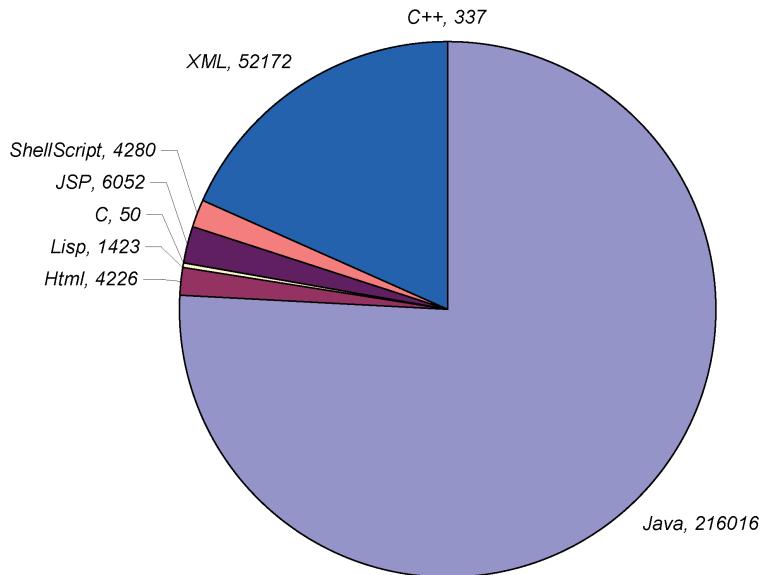


Figure 7.1. Hackystat Size by Programming Language

event to their assignments. An automated integration build tool was used to handle the full system build and test. Every night, the tool checked out the latest revision of the entire source, compiled, built, and tested the system. If there was any error, an email was sent to the development team. Figure 7.2 is a graphical illustration of the process followed by the developers in the lab.

Issues and project progress were tracked by an issue management system called *Jira*. The lab had a status meeting every week. Code review was conducted as needed, but not on a regular basis. Though the team did not treat the software development as a strict *timebox*,² it made regular releases about every three months.

In order to increase the development team's awareness of software metrics, the CSDL development environment was instrumented with sensors to collect a variety of software product and process metrics. These metrics were sent to a server in CSDL for storage and analysis. I wrote a client-side application that automatically extracted telemetry charts from the server on a regular basis and displayed them on a 3x3 array of nine LCD monitors mounted on a wall inside the lab. Figure 7.3 is a picture of it. I call the nine-monitor wall the “*telemetry wall*,” and the client-side application the “*telemetry control center*.”

²In software project management, a *timebox* is a period of time in which to accomplish some task. The end date is set in stone and may not be changed. If necessary, less functionality than originally planned is provided on the release date.

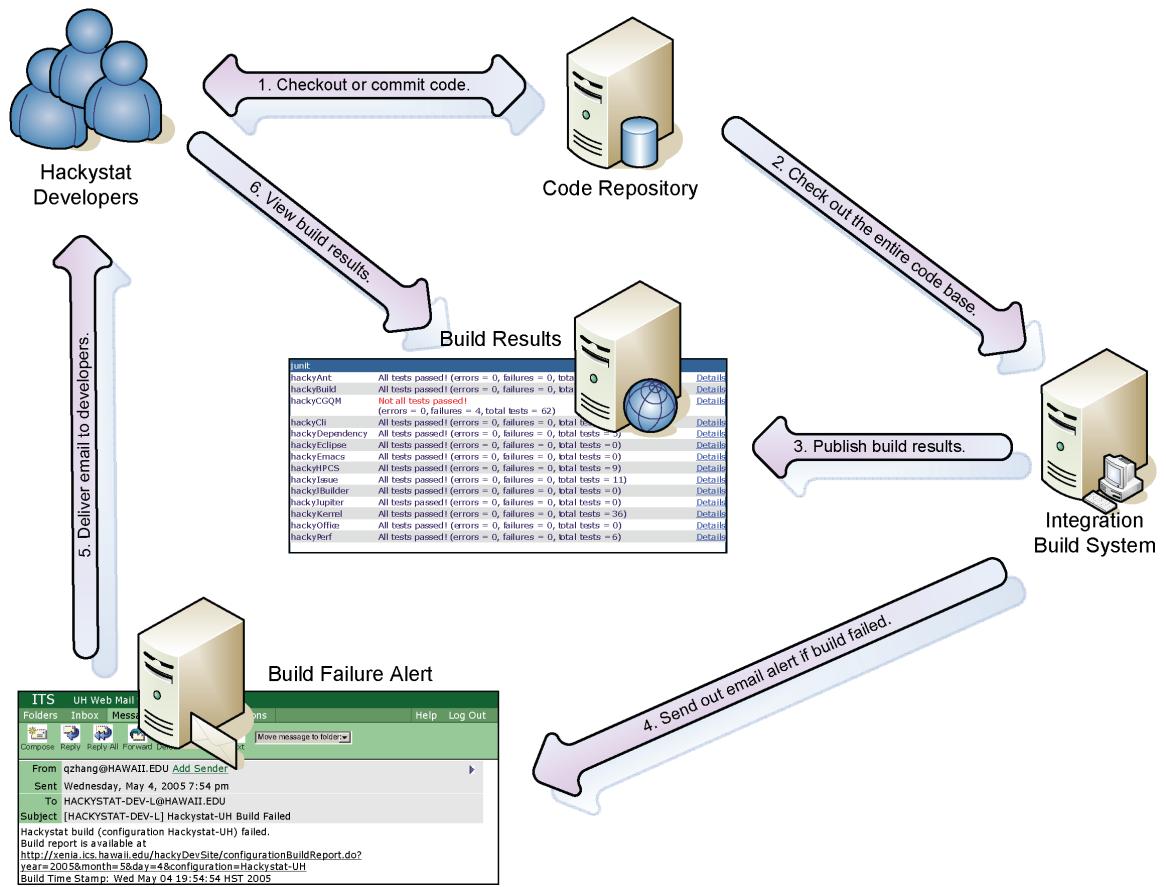


Figure 7.2. CSDL Software Development Process

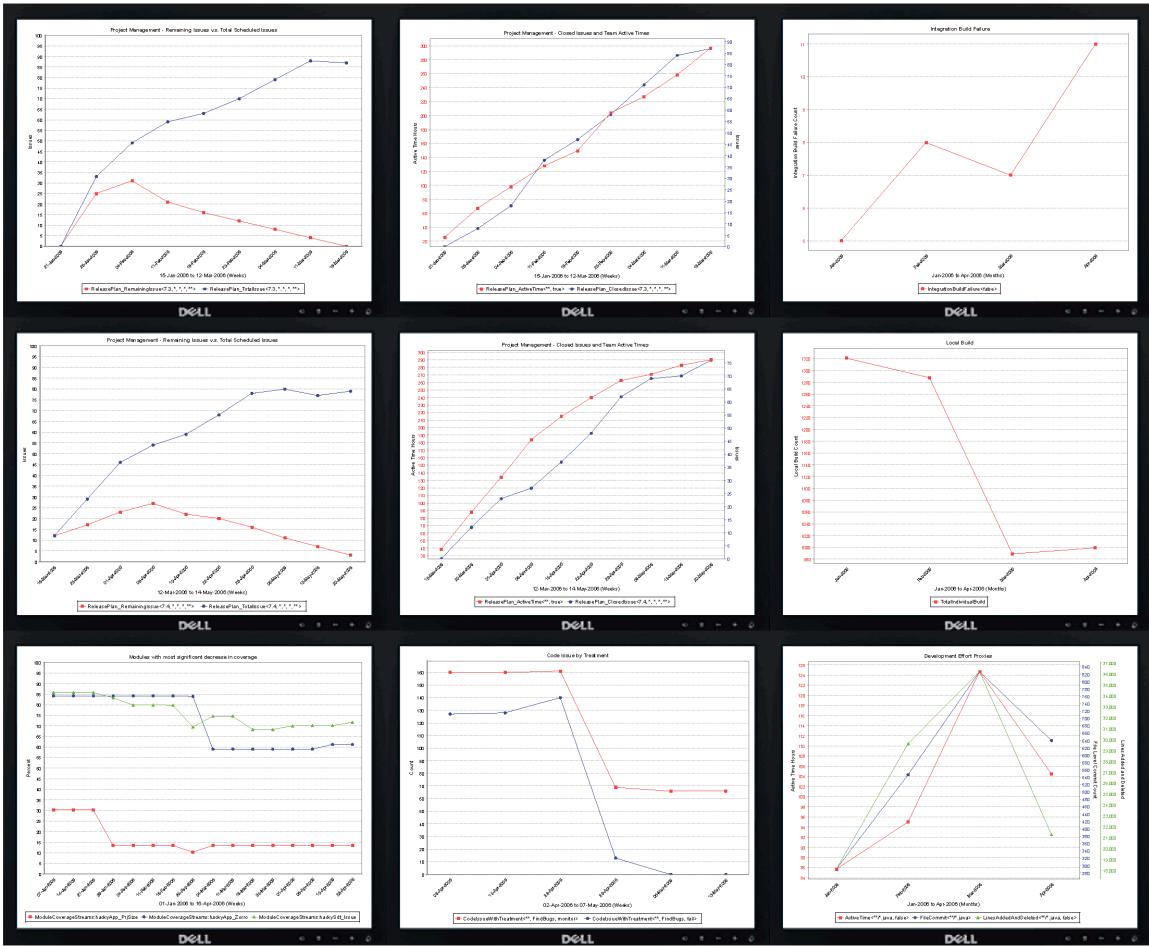


Figure 7.3. Telemetry Control Center on Telemetry Wall

A configuration file provided the definitions of the telemetry charts to be displayed using the standard telemetry language constructs (Section 3.3). The telemetry control center retrieved the charts through the telemetry expert analysis interface (Figure 1.3) provided by the software project telemetry implementation (Chapter 4). The charts on the nine monitors formed a “*telemetry scene*.” The telemetry control center application could be configured to cycle through different scenes automatically.

The telemetry wall made a sequence of telemetry charts continuously available to the entire team automatically without any action on the part of the developers or the project manager. Rather than having to wait for a status update meeting, they could simply look at the telemetry wall to get a perspective on the current state of development. The development team had access to the regular telemetry analysis interface (Figure 1.3 and 1.2) through a web browser as well. However, the primary way I used to communicate telemetry analysis results to the team was the telemetry wall. It made it easy to discuss telemetry charts either formally in CSDL meetings or casually during lunch hours.

7.2 Researcher’s Role

My affiliation with CSDL started in 2003. I implemented the software project telemetry system as an extension to Hackystat. Dr. Philip Johnson is the director of CSDL. He is my dissertation adviser. He is also the project manager of Hackystat. I received a cornucopia of helpful advice from him during my implementation of software project telemetry.

During the period of this study, I acted as an on-site process expert introducing software project telemetry as a metrics-based process improvement program. I took careful observation of the team’s development activity. I interviewed the team members and the project manager. I defined telemetry charts and made them available on the telemetry wall (Figure 7.3). I discussed telemetry analysis results with the developers and the project manager. I recommended process changes. I helped the project manager institute changes to improve project management practices. I also helped the developers gain insights into their development processes.

At the same time, my own software development effort was concentrated on improving software project telemetry implementation based on the feedback received in this study. It included: (1) making sensors and telemetry charts available to meet the team’s process-improvement and decision-

making requirements, (2) enhancing the telemetry language to improve the display of telemetry charts, and (3) profiling and eliminating telemetry analysis runtime performance bottlenecks.

7.3 Study Design

The CSDL study was a mix-methods study to explore the use of software project telemetry in depth. I have been affiliated with the lab for three years. Because of my familiarity with the CSDL software development environment and the small number of developers involved in the study, I was able to pursue a much more comprehensive data collection and analysis strategy over a much longer period of time compared to the classroom study. Instead of just giving the developers software project telemetry tools and observing how they used them, I took a more active role by acting as a process expert. I introduced software project telemetry as a metrics-based process improvement program. I proposed improvements to the CSDL software development process, and helped the project manager institute the changes. The steps I took consisted of the following iterative steps:

1. Collect data from observations and interviews.
2. Code the data in order to generate hypotheses.
3. Propose improvements to the CSDL software development process.
4. Continue to collect data to assess the effectiveness of the changes.
5. Draw conclusions from the data gathered.

The study involved many elements from the constructivist paradigm. It was a case study in which I explored the use of software project telemetry in CSDL extensively. I collected data from observations and interviews, and generated hypotheses from the data. My observation strategy was complete involvement as a full participant instead of an outside spectator. I was in the lab almost every day working with the developers, and attended every weekly status update meeting. My interview strategy was in-depth interview instead of structured interview. Both formal and informal interviews were used. I always encouraged free responses. The purpose was to gather much richer data about how the developers and the project manager interacted with software project telemetry to make decisions. I have to admit that my affiliation with CSDL was a source of bias. Years of affiliation might have ingrained in me the software process practiced in CSDL, and thus made me unable to see important information in my observations and interviews. However, the bias

was mitigated through several factors. My observation was much less disruptive to the developers than having a stranger watching over their backs. My detailed knowledge about the project under development made it easy for me to link the observed facts to the contextual information in CSDL, so that I could have much deeper insights into “what’s going on” than an outsider. I often reconciled my interpretation of observed facts with the developers and the project manager in interviews.

The study also involved an element from the post-positivist paradigm. After the hypotheses were generated, I tested them in a limited way by making changes to the telemetry system or implementing new facilities to see if the hypothesized outcome would come true. To some extent, this hypothesis testing procedure could be viewed as the simplest and uncontrolled form of experiment. The difference is that most experiments rely on statistical analysis to draw conclusions, but mine does not.

7.4 Data Collection and Analysis

My data came from both observations and interviews. The observations included almost everything related to CSDL software development. The interviews covered a variety of topics, from the discussion of the current development process and telemetry analysis results, to the exploration of improvement options and assessment of change impact.

The data were stored and organized using a software program called “*Confluence*.” The reason that I used the software was because of the ease with which different types of documents could be organized and accessed anywhere through a web interface. Though Confluence is mainly designed for knowledge sharing, I did not share my field notes with the study participants. I divided the *Confluence* data storage area into two sections: “Raw Data” and “Hypotheses.” At the end of the study, I had accumulated a total of 173 entries in the “Raw Data” section, and generated a total of 9 findings in the “Hypotheses” section.

- **Raw Data** — They were field notes from observations and informal interviews, and transcripts from formal interviews. The data in this category were further organized into two levels. Second level entries were immediate follow-up observations and interviews closely related to their parent entry in the first level. The total 173 entries in this section consisted of 109 first level entries and 64 second level entries. Figure 7.4 shows an index page with links to all raw data entries, while Figure 7.5 shows the details of one of the entries. The raw data

themselves cannot be published because of privacy issues. However, I included a summary for each first level entry in Appendix C. The summaries were assigned unique identification numbers, which were referenced in my discussion of the study results in Section 7.5. The intention is to provide the reader a mechanism to “audit” my conclusions in some sense.

- **Hypotheses** — They were hypotheses regarding the use of software project telemetry from my conceptualized ideas based on the raw data. All entries in this section were formatted into five subsections: (1) pre-hypothesis links to raw data entries, (2) generated hypothesis, (3) change implementation, (4) post-hypothesis links to raw data entries, and (5) conclusion. Figure 7.6 is an index page with links to all generated hypotheses, while Figure 7.7 shows the details of one of the hypotheses. The details of each finding were reported in Section 7.5.

The approach I followed to generate hypotheses from the data was inspired by grounded theory. As soon as the field notes and interview transcripts were entered into the raw data section, I applied “*open coding*” to label the text with different category names (i.e., “*conceptualization*” of “what’s going on” in grounded theory terminology). The coding was done as annotations to the raw data text (Figure 7.5), and was stored together with the raw data entry. I constantly compared, modified, and merged the category names when new data came in. During the process of *open coding*, some themes (i.e., “*core variables*”) emerged naturally. They were related to the use of software project telemetry. I continued to collect and categorize data after the emergence of the themes. However, at this step, my data collection was more selective. I paid more attention to those related to the identified themes (i.e., “*selective coding*”). The next step was “*theoretical memoing*”, in which I generated my hypotheses, such as the best practice of software project telemetry, the plausible reason for the problems encountered during its use, and the possible remedy to improve the system. The hypotheses were entered in the “*hypotheses*” section in the Confluence data store, together with the links to the relevant raw data entries. I did not have a separate “*sorting*” step. The links to the relevant raw data in each hypothesis entry served the same purpose as sorting. In contrast to grounded theory where hypotheses are the end goal, my data collection and analysis did not stop there. After the hypotheses were generated, I made changes to the telemetry system, implemented new facilities, and collected additional data in an attempt to confirm or refine the hypotheses. The additional data were recorded in the “*raw data*” section as well, and any refinement to the previously generated hypotheses was also noted in the “*hypotheses*” section.

Raw Observational Data - TelemetryCaseStudy - Confluence - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost/display/TCS/Raw+Observational+Data

Dashboard > TelemetryCaseStudy > Home > Raw Observational Data

Welcome QZhang | History | Profile | Log Out

Raw Observational Data

View Edit Attachments (0) Info

Added by QZhang, last edited by QZhang on Jul 12, 2006 (view change)

Labels: (None) EDIT

Raw Data:

- [2006-01-05-1 Interview with the Project Manager](#)
- [2006-01-05-2 Casual Chat with Developer 2 and 4](#)
- [2006-01-06-1 Interview with Developer 2](#)
- [2006-01-09-1 Interview with Developer 5](#)
- [2006-01-11-1 Interview with Developer 4](#)
- [2006-01-12-1 Build Failure Alert](#)
 - [2006-01-12-B Build Failure Analysis](#)
 - [2006-01-12-C Build Failure Cause from Developer 3](#)
 - [2006-01-13-B Casual Chat on Build Failure Impact with Developer 4](#)
 - [2006-01-15-A Build Failure Follow-up Email with Developer 3](#)
- [2006-01-16-1 CSDL Meeting Minutes](#)

Figure 7.4. A Page with Links to all Raw Data Entries

2006-02-06-1 CSDL Meeting, Intro Telemetry Wall, Discussion - TelemetryCaseStudy - Confluence - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost/display/TCS/2006-02-06-1+CSDL+Meeting%2C+Intro+Telemetry+Wall%

Dashboard > TelemetryCaseStudy > ... > Raw Observational Data > 2006-02-06-1

Welcome QZhang | History | Profile | Log Out

2006-02-06-1 CSDL Meeting, Intro Telemetry Wall, Discussion

View Edit Attachments (2) Info

Added by QZhang, last edited by QZhang on Jul 11, 2006 (view change)

Labels: (None) EDIT

Telemetry Design

Bottom-up telemetry scene design does not seem useful.

[Introduction of charts on Telemetry Wall, and soliciting comments]

[Me] I have put some scenes on the telemetry wall. What you can see from here... Basically, this is a short term overview, only for the last 14 days. This is the number of total issue we have for release 7.3. The number of total issue goes up. This one is the number of open issues. This one is integration build failure, a simple count, the number of integration build failure for the last 2 weeks. This one is the number of files committed each day. This one is active

Figure 7.5. One of the Raw Data Entries with Annotation

Id	Hypothesis
Finding-1	SPT improved CSDL issue management practice.
Finding-2	SPT helped improve CSDL code quality.
Finding-3	SPT provided the developers with insights into their processes.
Finding-4	Top-down telemetry design is a best practice.
Finding-5	Using SPT for sensor data self-validation is a best practice.
Finding-6	SPT will not likely to provide value when low-level details is required.
Finding-7	Filter functions is a useful enhancement to SPT.
Finding-8	Manually-specified y-axis is a useful feature to SPT.
Finding-9	SPT runtime performance improvement

Figure 7.6. A Tables with Links to all Generated Hypotheses

Pre-hypothesis Data:

- Ob-2006-01-16-1 CSDL weekly status meeting: the developers noted that one could generate a lot of telemetry charts using the telemetry language, but most of them were not that useful.
- Ob-2006-02-05-1 The telemetry wall was up. It was designed in a bottom-up fashion, and the charts were organized by the types of metrics.
- Ob-2006-02-06-1 CSDL weekly status meeting: the project manager and the developers found that most of the charts were not useful, except the release cycle issue tracking charts.

Generated Hypothesis:

Top-down telemetry design is a best practice.

Figure 7.7. One of the Generated Hypotheses

7.5 Results

The results of the study are the findings regarding the use of software project telemetry. Each finding is reported in its own sub-section, which is organized into the following parts:

- **Pre-hypothesis Data³** — Summary information about the data I collected that led to the generated hypothesis.
- **Generated Hypothesis** — The hypothesis, such as the best practice of software project telemetry, the plausible reasons for the problems encountered during its use, and the possible remedies to improve the system.
- **Intervention** — The changes I introduced based on the hypothesis in order to use software project telemetry more effectively, or overcome the problems encountered during its use.
- **Post-hypothesis Data⁴** — The additional data I collected after the changes were implemented, which were used to confirm or refine the generated hypothesis.
- **Conclusion** — The final finding and comments.

Finally, at the end of each report, I included an “elaboration” to provide much more detailed account of the finding.

³To preserve privacy, I used the word “he” when referring to individual developer throughout my report, even though there were both male and female participants in the study.

⁴Same as above.

7.5.1 Improvement on CSDL Release Cycle Issue Management

Pre-hypothesis Data:

- 2006-01-05-1: I discussed the existing analysis of issue metrics with the project manager in an interview. He did not utilize the metrics in project management, because the analysis was inadequate for release cycle planning and tracking.
- 2006-01-05-2: I interviewed two developers on their opinions about the utility of the metrics currently collected in the lab. One of them thought the issue related metrics were not that useful, because they were quite different from what he had anticipated.
- 2006-01-06-1: I discussed the current status of issue management with a developer, who told me that most of his development activities were not recorded in the issue database.
- 2006-01-09-1: I held a discussion with another developer, who estimated that only 20% - 30% of his development activities were tracked by the issue database. He also told me that he never followed the issue priority in resolving issues assigned to him.
- 2006-01-11-1: I held a discussion with yet another developer, who estimated that less than 15% of his development activities were tracked by the issue database. He told me that most of his issues were assigned through emails instead of the issue management system. He also told me that he did not understand how issue metrics were computed.

Generated Hypothesis:

There were two reasons why the issue metrics were not useful: (1) the issue tracking system severely under-represented the actual development effort; and (2) the issues scheduled for a release did not reflect the actual items that the team wanted to accomplish in that release cycle. If these two problems could be resolved, then the issue tracking telemetry charts could be used not only to track the progress in a release cycle, but also to make in-process predictions about the release schedule.

Intervention:

I identified the problem to the project manager and the developers, and explored options with them to make the issue tracking database more consistent with the actual development effort.

Post-hypothesis Data:

- 2006-01-16-1: The project manager discussed possible changes to improve the issue management practice with the developers in a weekly status meeting.
- 2006-01-19-2: As part of corrective measures, the project manager went through all issues and tagged them with realistic fix version numbers to get ready for the new release cycle (i.e., Release 7.3).
- 2006-01-19-3: As part of corrective measures, the project manager sent out an email requiring that all future commits should have issue Id in commit log comment field.
- 2006-01-23-2: This was the first time in my observation that the new issues assigned in the weekly status meeting were recorded in the issue management system.
- 2006-01-29-1: I enhanced the Jira sensor to collect missing information required for issue tracking telemetry analyses.
- 2006-02-03-1: I made the issue tracking charts available on the telemetry wall, and showed them to a developer. He commented that they could be used not only to track issue status but also to predict system release date.
- 2006-02-06-1: The project issue tracking charts were formally introduced in a CSDL meeting. The project manager commented that they were “highly useful.”
- 2006-03-13-1: I interviewed a developer. He confirmed that almost all his work was tracked by the issue tracking system now.
- 2006-03-15-2: I interviewed two more developers. They all confirmed that most of their work was tracked by the issue tracking system.
- 2006-03-20-2: The project manager reviewed the issue tracking chart after release 7.3 was finished, and reflected that the chart helped him determine whether more issues could be added to that release.
- 2006-04-07-3: The project manager was comparing release 7.4 issue tracking chart with the chart from the previous release cycle to make short term predictions.
- 2006-04-26-1: During an interview, the project manager told me that his project management skill had improved a lot with respect to release cycle issue tracking and planning.

Conclusion:

The additional data appears to confirm the hypothesis. The corrective measures were simple but effective. The issue tracking charts were found useful by the project manager for release cycle progress tracking and in-process prediction.

Elaboration:

CSDL used an issue management tool called “*Jira*” to record and track issues related to the *Hackystat* development. The types of the issues managed by *Jira* included feature requests, task assignments, and software bugs. CSDL had been using *Jira* and collecting issue related metrics for almost two years. During an interview at the beginning of this study, the project manager revealed that he did not utilize the issue metrics to plan and track issues in a release cycle. I also interviewed the developers asking them their opinions about the metrics being collected in CSDL. One of them identified the issue metrics as one of not-so-useful metrics. Two problems were identified in the discussions, both related to the way issues were managed and tracked:

- The issue management system significantly under-represented the actual development effort. For example, the percentage of issues tracked by *Jira* was less than 15% according to one developer. Another developer estimated this number between 20% to 30%. A lot of issues were assigned in emails or weekly status meetings, instead of through the issue management system.
- The open issues in the issue management system did not correspond to the expectation for the set of items to be accomplished in a release cycle. I raised this problem in an interview with the project manager. In his own words, when preparing for a stable release, he simply “*chucked the remaining unresolved issues over the fence into the next version.*”

My hypothesis was that the issue metrics were not useful precisely because of the two problems identified above. If the issue tracking database could be made consistent with the actual development effort, then the issue telemetry charts could be used not only to track the progress in a release cycle, but also to make in-process predictions about the release schedule.

After I identified the problems to the project manager and the developers, they took corrective measures quickly. On the manager side, the project manager went over all the issues, prioritized

them, and tagged them with realistic fix version numbers. On the developer side, the developers were required to supply a log message referencing an issue number whenever making a commit. The idea was that the practice would remind them to create a *Jira* issue if it had not already existed.

The result of these changes was both effective and positive, because they made it possible to generate issue tracking charts that reflected the actual development activities. For example, Figure 7.8 and 7.9 were taken from the Hackystat release cycle 7.3, which lasted from mid-January to mid-March.

Figure 7.8 is a telemetry chart showing the number of total *vs.* remaining issues on the last day of each week during the Hackystat 7.3 release cycle. The blue line on the top is the total number of issues scheduled for that release, while the red line below is the number of remaining issues. The telemetry chart clearly indicates that CSDL did not schedule everything up-front but rather added new issues almost every week. There was a concern that adding new issues constantly might lead to unmanageable release cycles. However, the project manager told me that he used the trend about issue closure in the red line to control whether or not more issues could be added to the release cycle. As the chart indicates, after initial weeks of issue build-up, CSDL was able to make consistent progress toward zero open issue and the delivery of the 7.3 stable release.

Figure 7.9 is a telemetry chart showing the cumulative number of closed issues *vs.* the cumulative amount of developer “*active time*” on a weekly basis during the Hackystat 7.3 release cycle. Though the active time required for an individual issue varied significantly with the actual issue in question, over time these differences appeared to “*smooth out*.” The chart indicates that for the 7.3 release cycle, CSDL issue closure rate was pretty constant (around 12 issues per week, or 3 active time hours per issue). The near linear relationship is quite provocative: if this same relationship holds true for future releases, then it provides strong evidence for a predictive relationship and a basis for cost and schedule estimation.

Figure 7.10 and 7.11 were taken from the Hackystat release cycle 7.4. The telemetry exhibited similar trends as those in the release cycle 7.3. A revealing observation was that in the middle of the release cycle 7.4, the project manager was constantly comparing telemetry shapes between 7.3 and 7.4 releases and making in-process schedule predictions about 7.4 release date. These charts not only allowed CSDL to track progress of each release cycle, but also established a baseline for planning and scheduling of future releases.

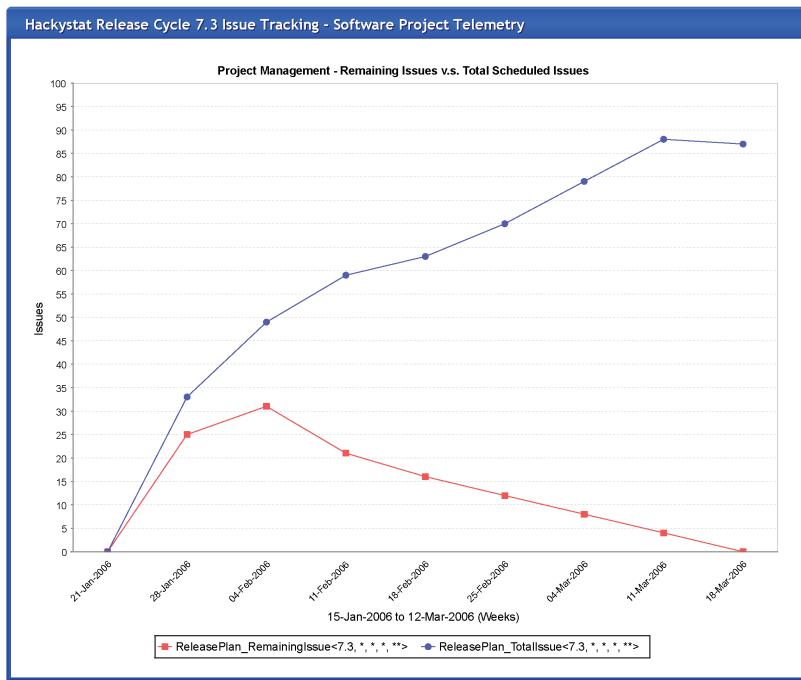


Figure 7.8. Hackystat Release Cycle 7.3 — Total Issues vs. Remaining Issues

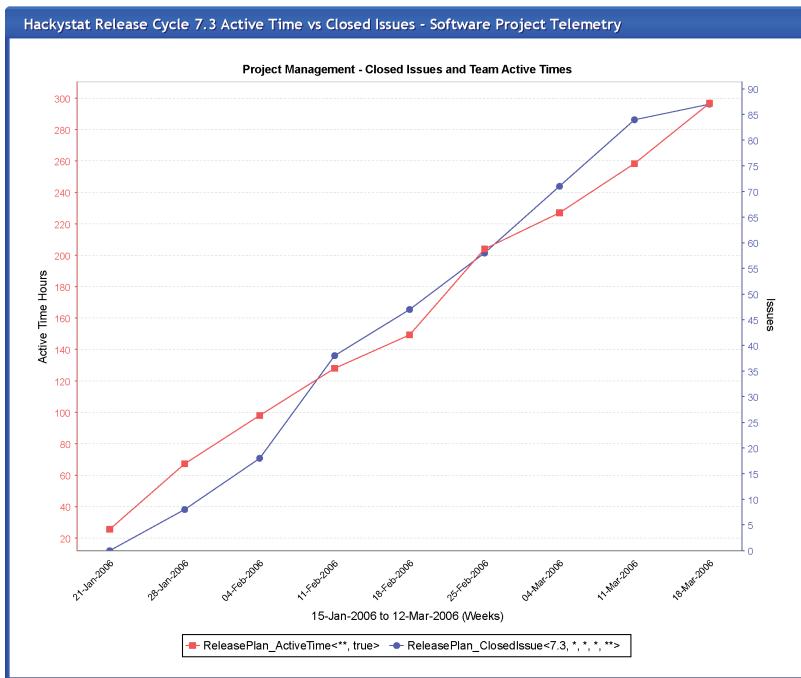


Figure 7.9. Hackystat Release Cycle 7.3 — Total Issues vs. Active Time

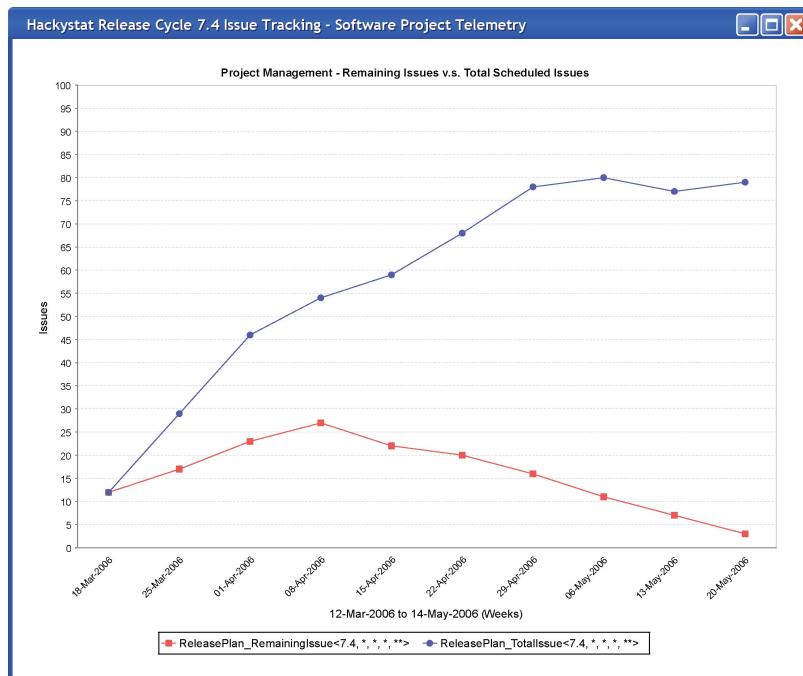


Figure 7.10. Hackystat Release Cycle 7.4 — Total Issues vs. Remaining Issues

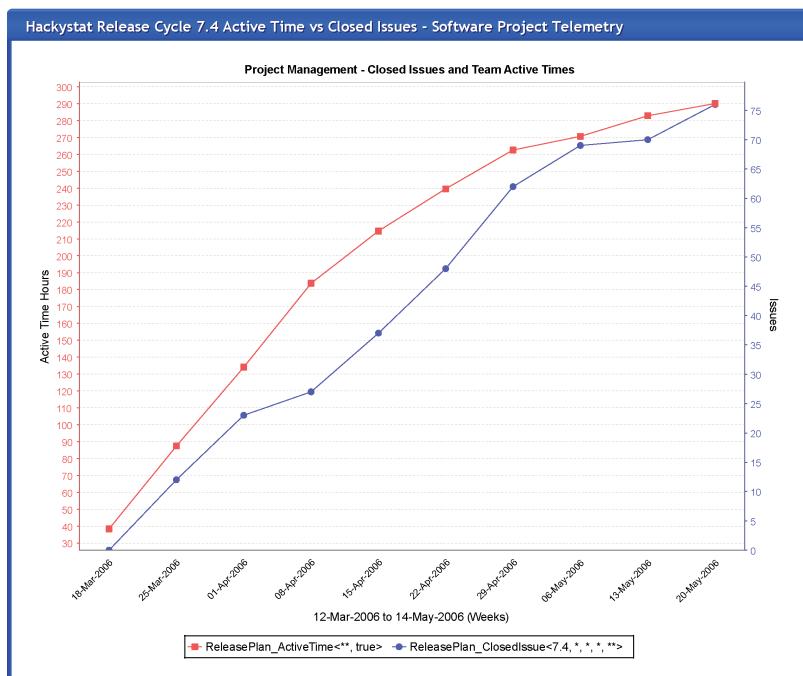


Figure 7.11. Hackystat Release Cycle 7.4 — Total Issues vs. Active Time

7.5.2 Improvement on CSDL Code Quality

Pre-hypothesis Data:

- 2006-01-09-1: I asked a developer about his perception of the utility of the metrics currently collected in the lab during an interview. He told me that FindBugs reported too many problems, and that a lot of them were false-positive.
- 2006-01-11-1: I asked another developer about his perceptions about the utility of the metrics currently collected in the lab. He told me that metrics from FindBugs and PMD were not useful, but they could be made useful if the false-positive problem could be resolved.
- 2006-01-16-1: In the weekly status meeting, the developers commented on the metrics from FindBugs and PMD. They all appeared to agree that there were too many false-positive warnings.
- 2006-02-09-2: CSDL deployed FindBugs and PMD sensors, even though the false-positive issue had not been addressed.
- 2006-03-23-1: I made the code issue density charts, which were computed from FindBugs and PMD metrics, available on the telemetry wall. I showed the charts to two of the developers. They told me that the charts failed to provide clue about Hackystat code quality, because they did not know the rules used by FindBugs and PMD to generate warnings.

Generated Hypothesis:

The FindBugs and PMD warnings would be useful in improving Hackystat code quality, if the false-positive problem could be resolved. Since the development team in CSDL did not have enough resource to examine every single warning, the key to benefit from the tools was to prioritize the warnings they produced. Assuming different types of warnings had different probability of being false-positive, this probability could be used to determine the priority of the warnings.

Intervention:

I started with FindBug warnings and categorized them into three groups with different treatment options: *fail*, *monitor*, and *ignore*.

Post-hypothesis Data:

- 2006-04-17-3: I discussed with the developers treatment options for each of the 17 types of FindBugs warnings found in `hackyCore_Kernel` module in the weekly status meeting. The comments from the developers indicated that they had learned a lot by going over their own code that generated the warnings.
- 2006-04-24-1: I discussed with the developers treatment options for the remaining types of FindBugs warnings found in the `Hackystat` source in the weekly status meeting.
- 2006-04-25-1: I modified the code issue telemetry chart to track the number of warnings falling into “fail” and “monitor” categories.
- 2006-04-26-2: The project manager assigned tasks for the developers to get rid of the warnings that fell into the “fail” category.
- 2006-05-06-1: Telemetry analysis indicated that all the FindBugs warnings in the “fail” category had been eliminated, and the warnings in the “monitor” category had been reduced by more than a half.

Conclusion:

The additional data appears to confirm the hypothesis. The intervention was successful. Within two weeks, the developers had completely eliminated the FindBugs warnings in the “*fail*” category, and drove down the FindBugs warnings in the “*monitor*” category by more than a half.

Elaboration:

The “*CodeIssue*” metric is a type of metric designed for representation of problems uncovered by static code analysis tools such as FindBugs[31] and PMD[69]. These tools perform static analysis either on Java source code or compiled byte code, and flag suspicious language constructs, which are potential software bugs. For example, a compiler won’t complain if you try to dereference a null pointer, but when executing the code the most probable outcome is application crash.

In February 2006, CSDL deployed both FindBugs and PMD to run on the `Hackystat` source. Unlike a compiler where the distinction between bug and non-bug is clear, these static code analyzers can only make probability statements about potential bugs. The general feeling among the developers was that there were too many warnings reported by the tools, and that most of them were

false-positive. The telemetry chart in Figure 7.12 indicated that in the nine weeks from Feb 25 to April 22, the number of warnings reported by FindBugs had increased from 507 to 518, while the number reported by PMD had increased from 6167 to 6702. The trends looked bad, but nobody was sure whether they constituted proof that the quality of the project had indeed gone down. One developer commented: “*These numbers are potentially useful, but I don’t think there are really useful at this time. There are so many false-positives.*”

My hypothesis was that the warnings reported by FindBugs and PMD contained valuable information to improve Hackystat code quality. Since the development team in CSDL did not have enough resource to examine every single warning, the key to benefit from the tools was to prioritize the warnings they produced. Assuming different types of warnings had different probability of being false-positive, this probability could be used to determine the priority of the warnings. Based on this hypothesis, I decided to put them into three categories with different treatment options:

- **Fail** — These were the types of warnings with high probability of being true. The existence of such warnings should fail CSDL nightly integration build, so that they could be eliminated immediately after they were detected.
- **Monitor** — These were the types of warnings with moderate probability of being true. They did not have to be dealt with immediately given the resource constraint faced by CSDL development team. However, the number of these types of warnings should be used as quality indicator and closely monitored for bad trend.
- **Ignore** — These were the types of warnings with low probability of being true. The development team should not waste any resource on them.

I started with FindBug warnings. For each type of warning, I picked one instance and discussed treatment options with the developers in CSDL weekly meetings. The discussion was overwhelmingly welcomed. The developers told me that they learned a lot about best coding practices to avoid common errors with the samples of warnings generated from their own code.

Figure 7.13 is a telemetry chart showing the number of FindBugs warnings that fell into “*fail*” and “*monitor*” categories respectively. The chart indicated that within two weeks after the discussion, the developers had completely eliminated the warnings in the “*fail*” category, and drove down the number of warnings in the “*monitor*” category by more than a half. The project manager was

so happy with the results, that he told me he would follow my approach to do the same thing with PMD warnings after the summer.

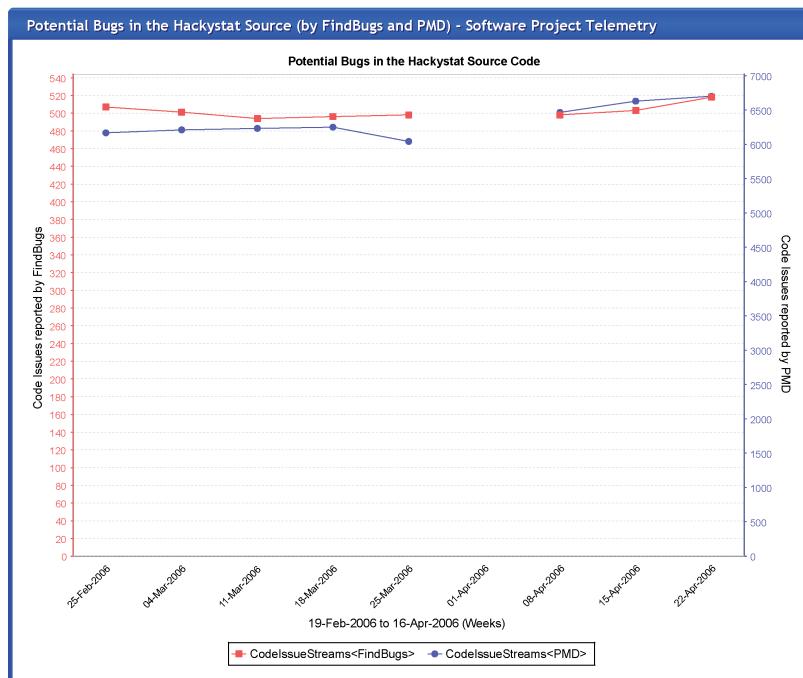


Figure 7.12. FindBugs and PMD Warnings from the Hackystat Source

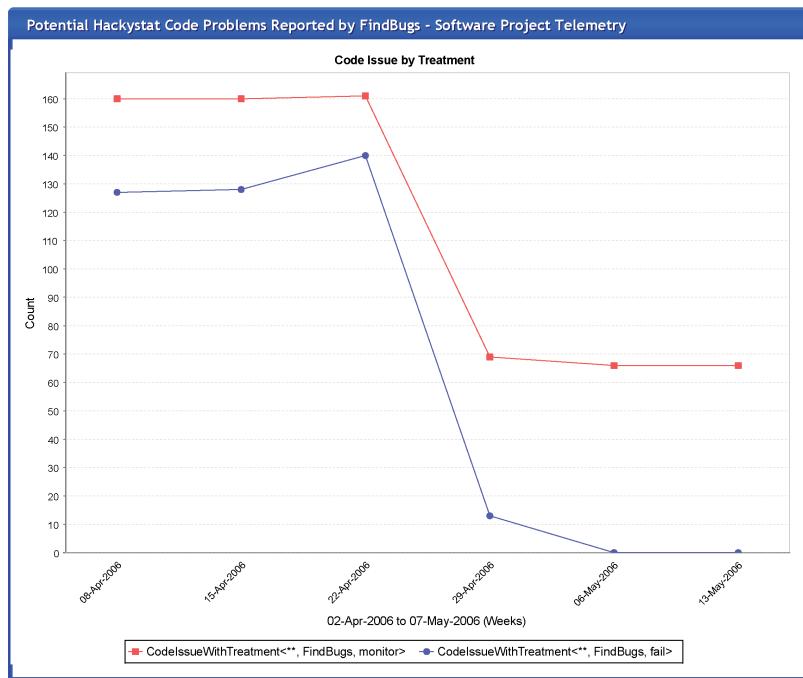


Figure 7.13. FindBugs Warnings in “Fail” and “Monitor” Categories

7.5.3 Improvement on Developers' Insights into their Software Development Process

Pre-hypothesis Data:

The pre-hypothesis data came from a previous study in which I analyzed 2004 CSDL integration build failure data. The study found that the build failure rate was significant. 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. The study suggested that the causes of the integration build failures were quite complex. They involved at least the following: developer's familiarity with the system, the actual changes made to the code, the dependency relationships among the modules. Since there was no statistical correlation between integration build failures and the number of lines of code committed or the amount of active time spent before the commit, it would be difficult to adopt the traditional approach of building an analytical model to predict the probability of integration build failure in order to forewarn the developers. At the same time, the study also suggested that 74% – 82% of the integration build failures were preventable if the developers could build and test the system on their workstations before committing the changes to the repository. However, there is a dilemma. On the one hand, the developers often do not test their changes against the entire code base before committing them, because a full build and test could take over 15-20 minutes, which would be quite time-consuming given that the developers often commit more than once a day. On the other hand, the cost of a broken build is quite time-consuming too, since it could prevent other developers from working when the code repository is in an inconsistent state.

Generated Hypothesis:

Software project telemetry could be used to provide feedback to the developers to help them gain insights into their software development processes and the cost associated with integration build failures, so that they could learn from their past experiences to test “just the right amount” of the system before committing their changes, where “just the right amount” involves a trade-off between local quality assurance effort and increased risk of an integration build failure.

Intervention:

Software project telemetry was used to provide process feedback to the developers. To draw their attention, I implemented an email alert that automatically identified the plausible developer responsible for the build failure whenever possible.

Post-hypothesis Data:

- 2006-01-19-2: I interviewed a developer (developer 1) who was identified as responsible for a recent integration build failure. I asked him how it might impact his local quality assurance practice. He told me that it was “*very effective*” in making him think more about the integration build result when committing changes.
- 2006-01-31-2: I interviewed a developer (developer 2) on the impact of the integration build failure alert mechanism. He commented that it made him “*a little bit more cautious*” when committing changes.
- 2006-01-31-3: I interviewed developer 1 on the impact of the integration build failure email alert mechanism. He commented: “*you might be identified as a culprit, (which) tends to make you try a little harder to think about when you actually do the thing (i.e., committing changes).*”
- 2006-02-02-3: I interviewed a developer (developer 3) on the impact of the integration build failure alert mechanism. He told me that his behavior was changed significantly from “*I just build the module to see if it works*” to “*I build the entire system and test every time before commit.*”
- 2006-02-02-4: I interviewed a developer (developer 4) on the impact of the integration build failure alert mechanism. He told me that it had not changed his behavior because he was always careful about local quality assurance.
- 2006-03-08-1: I interviewed a developer (developer 5) on the impact of the integration build failure alert mechanism. He told me that he had spent more time on local quality assurance than before. There was overhead, but it was acceptable since it would be much more troublesome to have integration build failures.
- 2006-03-08-2: I interviewed developer 4. He controlled the local quality assurance overhead by reducing the number of commits.

- 2006-03-14-2: I interviewed developer 2. He commented that the overhead on local quality assurance was acceptable given the consequence of integration build failures. He controlled the overhead by reducing the number of commits.
- 2006-03-23-1: I discussed the telemetry charts on integration build failures and various software development process metrics with two of the developers. They confirmed that integration build failure was a complex phenomenon, and that it would be very hard, if not impossible, to predict the probability from the process metrics.

Conclusion:

The post-hypothesis data appears to confirm the hypothesis. By providing process feedback to the developers, software project telemetry seemed to make them more aware of the cost associated with integration build failures and thus more careful when committing their changes. Analysis of the developer's process metrics and integration build failure trends (Figure 7.14) seemed to confirm that they were learning from their past experiences and getting "*smarter*" about their local quality assurance practices.

Elaboration:

The Hackystat project is so large that the developers usually work on a subset of the modules relevant to their assignments. An automated integration build tool is used in CSDL to build and test the entire code to make sure that the developers' modifications do not break the system. The process is illustrated in Figure 7.2 and discussed in Section 7.1. In early 2005, I conducted a study on CSDL integration build failures using 2004 data. The study suggested that the causes of the integration build failures were quite complex, which involved many factors, such as developer's familiarity with the system, the actual changes made to the code, the dependency relationships among the modules, etc. As a result, I was unable to find statistical correlation that could be used to describe the causal relationship between software development practices and integration build failures in CSDL. However, the study did suggest there was a trade-off between developers' local quality assurance effort and integration build failures. Testing the changes against the entire code base before committing them could reduce the integration build failure rate significantly, but it was quite time-consuming. On the other hand, a failed integration build would often waste other developer's time because the code in the repository was in an unsafe state.

My hypothesis in this study was that the phenomena of software development practices and integration build failures are so complex that it would be futile to build a predictive model to describe the causal relationship between them. Instead, software project telemetry could be used to provide feedback to the developers to help them gain insights into their software development processes and the cost associated with integration build failures, so that they could learn from their past experiences to test “just the right amount” of the system before committing their changes, where “just the right amount” involves a trade-off between local quality assurance effort and increased risk of an integration build failure. In order to draw their attention to telemetry analysis results, I implemented an email alert that automatically identified the plausible developer responsible for the build failure whenever possible.

The data in this study indicated that the result was positive. The responses from the developers (see post-hypothesis data) seemed to suggest that software project telemetry made them more aware of the cost associated with integration build failures. As a result, they seemed to be more careful about committing their changes, and more effort was spent on local quality assurance.

The phenomena resembled the typical textbook case of game theory in Economics. Consider a hypothetical game Adam and Bob are playing, in which each player has two choices: choice 1 and 2. The payoff matrix is listed in Table 7.1. Now consider what would happen if Adam and Bob cannot communicate with each other. If Adam’s choice is 1, then Bob’s best response is 2 because he can get \$110 by choosing 2 instead of \$100 by choosing 1. On the other hand, if Adam’s choice is 2, then Bob’s best response is still 2 because he can get \$10 by choosing 2 instead of \$0 by choosing 1. In other words, Bob’s best response is always 2 regardless of Adam’s choice. If you apply the same logic to Adam, then you would find that Adam’s best response is always 2 regardless of Bob’s choice. Therefore, both Adam and Bob would end up with \$10 when they cannot communicate with each other, and thus cannot reach a mutually beneficial deal.⁵ Both players act rationally trying to maximize his own benefit. However, obviously, rational decisions are not always optimal, because if Adam and Bob could reach a binding agreement, then both of them would end up much better off by getting \$100 each.

Put it in the context of CSDL, I can hypothesize that the developers were always making rational decisions to reduce their software development effort. Before the introduction of software project telemetry, they were less aware of the productivity loss of the integration build failures incurred

⁵This is what is called Nash equilibrium in Economics. A much more complex version of it is often used to model market outcome in a duopoly competition.

Table 7.1. Nash Equilibrium in a Non-Cooperative Game

	Bob Choosing 1	Bob Choosing 2
Adam Choosing 1	<ul style="list-style-type: none"> – Adam gets \$100; – Bob gets \$100. 	<ul style="list-style-type: none"> – Adam gets \$0 ; – Bob gets \$110.
Adam Choosing 2	<ul style="list-style-type: none"> – Adam gets \$110; – Bob gets \$0. 	<ul style="list-style-type: none"> – Adam gets \$10; – Bob gets \$10.

by other developers, and their decisions were more or less focused on minimizing local quality assurance effort. As a result, they ended up in the low payoff position. On the other hand, the process feedback mechanism I introduced with software project telemetry made them more aware of the cost associated with the integration build failures, and they began to make trade-off decisions to minimize the total combined cost instead of only local quality assurance cost. As a result, though the local quality assurance effort as experienced by individual developers had increased, the entire team were actually moving toward the high payoff position.

Figure 7.14 provides evidence that the developers were learning to make trade-off decisions between reducing local quality assurance cost and reducing integration build failure cost in order to minimize the total cost. The first chart shows the number of integration build failures in each month from January to April 2006. The second chart shows the number of times that the build script was invoked by the developers on their workstations. A typical purpose was to test the modifications locally before committing them to the repository. The last chart showed three telemetry streams on *FileCommit*, *CodeChurn*, and *ActiveTime*. *FileCommit* measures the total number of files in all commits from all developers. *CodeChurn* is related to file commit. It computes the total number of lines added and deleted in each revision. *ActiveTime* computes the amount of time a developer spent actively editing code inside an IDE. They all measure software development effort except from different angles.

It seemed that the developers were learning from their past experiences and getting “smarter” about their processes. January was the month with low development effort, large amount of local testing, and low integration build failures. It seemed to suggest that low integration build failure was achieved at the cost of lots of local tests. In February, development effort went up with no significant change in local testing pattern, resulting in higher number of integration build failures. March was the month with high development effort, moderate integration build failure rate, and very low local testing effort. This seemed to support the hypothesis that the developers had learned from their past experiences to test “just the right amount” of the system before committing their changes.

On a cursory look, the April data seemed contradictory to that hypothesis, with extremely high rate of integration build failure. But further investigation indicated that the April data were not directly comparable. April was the time that CSDL embarked on a completely different type of software development task: the team were busy with updating the Hackystat infrastructure code to support sensor data type evolution. The integration build failures concentrated on several non-actively-maintained leaf modules that were not in the developers' working set. They were exactly the type of build failures the integration build system was designed to detect. My CSDL study ended in early May, but I can conjecture that with continued metrics collection, I would find evidence that the developers are beginning to learn to deal with the new situation and bringing integration build failure back under control.

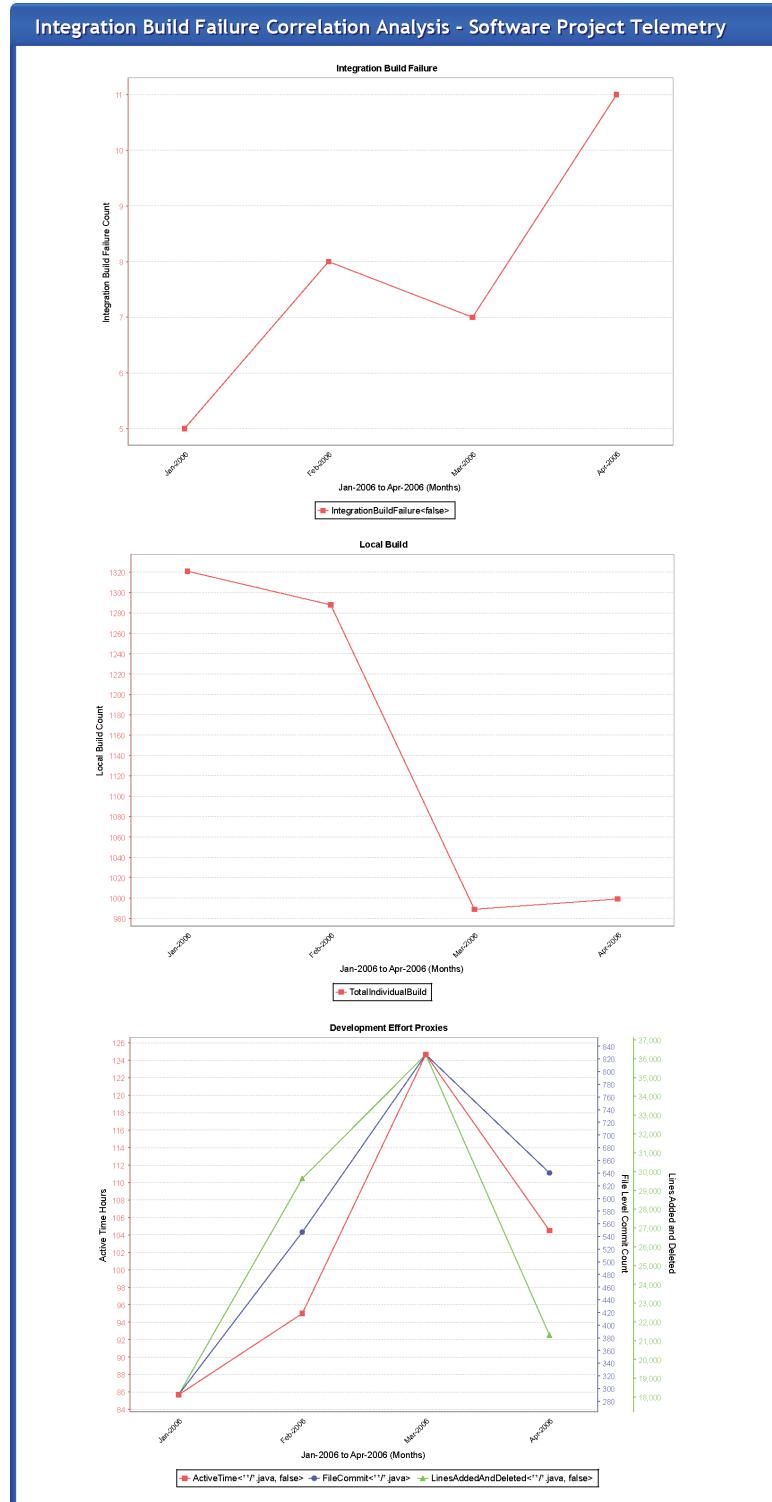


Figure 7.14. Integration Build Failures and Process Metrics

7.5.4 Top-down Telemetry Design

Pre-hypothesis Data:

- 2006-01-16-1: In a weekly status meeting, the developers noted that one could generate a lot of telemetry charts using the telemetry language. But they also asked: “*do we really care about all those charts?*”
- 2006-02-05-1: I gathered a list of the types of metrics collected in CSDL, and generated telemetry charts to show how these metrics changed over time in different grain size. The telemetry wall was up.
- 2006-02-06-1: I introduced the telemetry wall in the weekly status meeting, but the project manager and the developers found that most of the charts were not useful, except the release cycle issue tracking charts. They gave me a list of questions that they wished telemetry charts could help shed light on, such as some notion of quality indicators for each module in the project.

Generated Hypothesis:

The telemetry charts generated in a bottom-up fashion (i.e., organized by the types of metrics) were generally not useful, because they were not designed with any purpose in mind. You cannot expect the project manager or the developers to fish around hundreds of charts to find the ones that are useful to them. The reason that the issue tracking charts were found useful was because they happened to have a purpose: allowing the manager to track progress in a release cycle. Therefore, useful telemetry charts are those that are designed with a specific top-level goal in mind.

Intervention:

I redesigned the telemetry charts for the telemetry wall, organizing them by their intended use (i.e., top-down design). For example, there was a scene (a set of related charts displayed together on the telemetry wall) for release cycle issue tracking, and there was another scene for module level quality indication.

Post-hypothesis Data:

- 2006-02-21-2: I discussed the top-down designed charts with three of the developers, and they thought the charts displayed useful information.
- 2006-02-22-1: I showed the charts to the project manager, and he liked them.
- 2006-03-07-1: After receiving complaints about missing coverage data, the project manager sent me an email asking whether it would be possible to design charts to help detect sensor malfunction.
- 2006-03-09-1: A set of charts specifically designed for the purpose of verifying developer-side process metrics were deployed on the telemetry wall. Immediately, I noticed that one of the developers had missing data. It turned out that the developer reinstalled the IDE but forgot to reattach the sensor.
- 2006-03-17-1: The project manager was so impressed with the utility of those top-down designed telemetry charts on the telemetry wall, that he decided to devote an entire page on the Hackystat website to publish the results.

Conclusion:

The additional data appears to confirm the hypothesis. Though “*bottom-up telemetry design*” based on the types of metrics can generate hundreds of charts without significant effort, the charts lack clear purposes and are generally of little value to users. “*Top-down telemetry design*” based on user goals, which is similar to the idea in the Goal-Question-Metric paradigm, yields useful telemetry charts.

Elaboration:

At the beginning of this study, I used bottom-up approach to generate telemetry charts to be displayed on the telemetry wall. I gathered a list of the types of metrics collected in CSDL, and then generated charts to show how these metrics change over time in different grain sizes with possible breakdown to individual developer or individual source code module. The charts were organized by the types of metrics. For example, a scene on the telemetry wall might be displaying charts all related to *active time*, and another scene might be displaying charts all related to *code issue* density.

With the bottom-up approach, hundreds of charts could be easily generated. I introduced them in a weekly status meeting. However, I found that most of them were not useful. A frequent question the project manager and the developers asked during my presentation was: “*Why would that be of use to me?*” The only exception was the charts related to release cycle issue tracking, which the project manager found “*highly useful.*” Further discussion revealed that the project manager and the developers had a number of questions they wished software project telemetry could shed light on, such as:

- “*Could telemetry help me identify the situation where the sensors are likely not being installed?*”
- “*Could telemetry offer some notion of quality so that it helps me locate the modules with low quality, or the modules that are changing with respect to quality?*”
- “*Could telemetry give me a sense of how we are making progress toward the stable release?*”

My hypothesis was that bottom-up telemetry design was unsuccessful because it generated hundreds of charts organized by metrics types. You just cannot expect users to go over all those charts to find the ones that are useful to them. The issue tracking charts were useful because they happened to serve a purpose: they enabled the project manager to track progress in a release cycle. In order for other charts to be useful, they have to be re-organized in a top-down fashion by the questions they intended to answer. Based on this hypothesis, I redesigned the telemetry charts following top-down approach, which resulted in a number of successful telemetry scenes being displayed on the telemetry wall:

- A telemetry scene for release cycle issue tracking.
- A telemetry scene for product metrics at project level for quality indication.
- A set of telemetry scenes for product metrics at module level for quality indication. One scene per module.
- A telemetry scene for module filtering to discover interesting information in a large project like *Hackystat*.
- A telemetry scene for software development process metrics and correlation analysis.
- A telemetry scene for sensor data verification.

The top-down designed telemetry charts were so successful that the project manager decided to devoted an entire page on Hackystat website to publish the results. A snapshot is captured in Figure 7.15 and 7.16. The URL of the web page is:

<http://www.hackystat.org/hackyDevSite/telemetryReport.do>

The web page contains a collection of both *historical* and *real-time* charts, which serves two purposes: (1) enabling the entire Hackystat developer community, especially those off-site developers, to monitor the project development status, and (2) demonstrating the power of the Hackystat framework by showing the achievement of one of its extensions.

Telemetry Report - Page 1



Developer Services

Telemetry Report

[Home](#) | [Public Server](#) | [Stable Release](#) | [Last Build](#) | [Build Archive](#) | [Telemetry Report](#) | [Defect Tracking](#) | [People](#) | [About](#)

One of the analysis systems we have developed with Hackystat is called "Software Project Telemetry". The basic idea is to take the process and product measures gathered for a project, and display them as trends over time at the grain size of days, weeks, or months. While this idea is simple in theory, in practice it is quite challenging to design a usable, extensible, and expressive mechanism for display and interpretation of process and product metric trends. Software Project Telemetry addresses this design problem through a sophisticated declarative language for defining telemetry charts, along with a "plugin" architecture to allow support for new process and product metrics.

Software Project Telemetry can support project decision making in a variety of ways. First, it can help discover "baseline" values in your projects: typical values for process or product measures that can aid in estimation or help discover the impact of process/product/tool changes on your development practices. Second, telemetry can reveal the occurrence of "anomalous" conditions: a sudden spike (or valley) in a process or product measure can indicate that the development conditions have changed in some way. Third, telemetry can help reveal "co-variance" in process or product measures. For example, it might show that when code churn exceeds some threshold value, the frequency of daily build failures increases. For details on the principles and practices of Software Project Telemetry, consult the [Hackystat User Guide](#) chapter or our research publications linked to the [home page](#) of this site.

At the Collaborative Software Development Laboratory, we have implemented a [Telemetry Wall](#) consisting of a nine-headed PC controlled by "Telemetry Control Center" (TCC) software that provides a rotating set of telemetry charts to developers.

While the TCC is useful for developers physically located in the lab, we have decided to make a subset of these charts available on this page as a service to external developers. In addition, the charts on this page are annotated to provide people unfamiliar with Hackystat and Software Project Telemetry some examples of how Software Project Telemetry is currently being used on an active development project.

The following sections show how Software Project Telemetry is applied to a variety of project management issues, including Release Planning and Tracking, Project Quality, Active Module Detection, Integration Build Failures, and so forth. In each section, we provide both a static "historical examples" of telemetry charts that illustrate an interesting phenomenon revealed by telemetry, as well as "real time data", which is one or more charts representing an analysis of our current project metrics on a daily basis. These sections are:

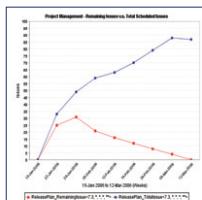
- [1. Release Cycle Issue Tracking](#)
- [2. Project Level Product Metrics and Quality Indicators](#)
- [3. Module Level Product Metrics and Quality Indicators](#)
- [4. Module Filtering for Large Project](#)
- [5. Development Process and Correlation Analysis](#)
- [6. Metrics Validation](#)

1. Release Cycle Issue Tracking

We use Issue Tracking telemetry to track trends in the open and closed issues over a specific release. This helps us manage progress toward the delivery of a stable release of Hackystat and helps us to identify schedule slippage.

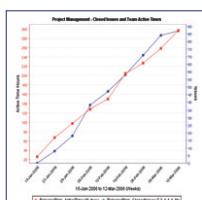
Historical Example:

The following example is taken from the Hackystat version 7.3 release cycle, which lasted from mid-January to mid-March.



Issue Tracking (Version 7.3):

This telemetry chart shows the total and remaining issues on the last day of each week during the 7.3 release cycle. The line on the top indicates the total number of issues scheduled for that release, while the line at the bottom indicates the number of remaining issues. The telemetry shows that we did not schedule everything up-front, but added new issues almost every week. It also shows that during the last half of the release cycle, we were able to make consistent progress toward zero open issues, indicating delivery of the stable release.



Issue Closure (Version 7.3):

This telemetry chart shows the cumulative number of closed issues and the cumulative amount of developer "active time" on a weekly basis during the 7.3 release cycle. Though the active time required for an individual issue varies significantly with the actual issue in question, over time these differences appear to "smooth out". This telemetry indicates that for this release cycle, our issue closure rate is pretty constant (around 12 issues per week or 3 active time hours per issue). The near linear relationship in this release cycle is provocative: if this same relationship between cumulative active time and issue closure holds in future release cycles, it would provide evidence for a predictive relationship.

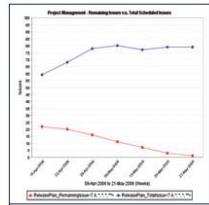
Figure 7.15. Telemetry Report: Page 1

Telemetry Report - Page 2

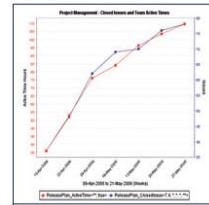
Real Time Data:

This section shows the Issue Tracking and Issue Closure telemetry charts for the (ongoing) Version 7.4 release cycle. Do these charts exhibit the same trends present in the Version 7.3 release cycle? When can we make the Version 7.4 release? Are we as productive as we were in the last release?

Issue Tracking (version 7.4)



Issue Closure (version 7.4)



2. Project Level Product Metrics and Quality Indicators

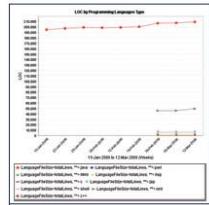
While the Release Planning telemetry might help us schedule our releases more accurately, what about the quality of our system? Does the schedule sacrifice quality?

The Hackstat project collects a wide variety of process and product measures related to quality, including: the rate of integration build failure, the frequency of unit test invocation, the success rate of unit tests, the coverage of unit tests, the presence of code review on a module, the number of active developers in a given module, and the density of issues detected by static code analyzers such as Checkstyle, PMD, and FindBugs. Any single measure provides only a small slice of insight into project quality, but when taken together and tracked over time, more meaningful and useful insights begin to emerge.

This section focuses on software product metrics, while [Section 5](#) focuses on process metrics. We use both types of metrics to guide our decision-making.

Historical Example:

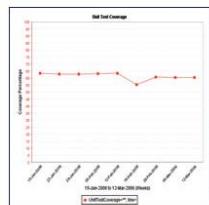
For the historical example, we show three charts: one for system size, the other two for two different perspectives on product quality. The charts are taken from the Hackstat version 7.3 release cycle.



Lines of Code (Version 7.3):

This chart shows source line of code for different types of language in the system. It clearly shows that the Hackstat is predominantly a Java system, but that it also includes code in several other languages, including C/C++, Perl, and even Lisp.

With each release, the system size increases. With Version 7.3 release, our Java code size broke 200K SLOC line.



Unit Test Coverage (Version 7.3):

This chart shows line level unit test coverage: the percentage of source code lines exercised by unit test cases. Only the coverage for Java code is shown.

It's interesting to note that though coverage telemetry is largely stable, it exhibits a slight downward trend. Understanding what this trend means requires diving down into the module-level coverages, as will be illustrated in [Section 4](#).

As a side note: The February 19 dip in coverage is an example of an "anomalous" data point. After investigation, we determined that metrics were not collected correctly on those days, causing the change. This also shows robustness of telemetry analysis: occasional incorrect metrics have little impact on the long-term value of telemetry charts. You can even use telemetry charts to ensure that your metrics are collected correctly, as illustrated in [Section 6](#) below.

Figure 7.16. Telemetry Report: Page 2

7.5.5 Sensor Verification

Pre-hypothesis Data:

- 2006-01-31-4: I noticed an inconsistency in telemetry charts: there was no data from one of the developers. It turned out it was caused by a bad server-side project configuration.
- 2006-02-01-1: The same developer told me he had fixed the problem, but the inconsistency still existed in telemetry charts. The project was still mis-configured despite the developer's effort to fix it.
- 2006-02-06-1: During my presentation of telemetry charts in the weekly status meeting, I noticed that the developers were using some of the charts to assess whether the underlying sensors data seemed correct or not. Further discussion identified two common causes for incorrect sensor data: (1) sensor not working correctly, and (2) bad server-side project configuration.

Generated Hypothesis:

There is always possibility for incorrect sensor data. Ensuring sensor data correctness is a tedious and time-consuming process, because a developer has to log onto the server to compare raw sensor data entries with his expectations. Specially designed telemetry charts could save much of the effort by allowing a developer to make quick assessment of the likelihood of occurrence of sensor data problem.

Intervention:

I designed a set of sensor data verification charts, and made them available both on the telemetry wall and on the public Hackystat website.

Post-hypothesis Data:

- 2006-02-26-1: Telemetry charts showed missing coverage data. It turned out that the sensor configuration file was not updated when a developer added a new module.
- 2006-03-05-1: Telemetry charts showed missing issue metrics. It turned out that a developer forgot to update the project configuration when adding a new module.

- 2006-03-09-1: I added additional telemetry charts on the telemetry wall, designed to verify developer-side process metrics. Immediately, I detected that one developer has missing active time data. It turned out that the developer reinstalled the IDE but forgot to reattach the sensor.
- 2006-04-20-1: A developer modified Jira sensor code. Telemetry charts showed missing issue metrics. It turned out it was caused by a bug in the code.
- 2006-04-22-2: An email from the project manager indicated he detected the same Jira sensor problem using the real-time sensor verification charts on the public Hackystat website.

Conclusion:

The data appears to suggest that it would be hard to avoid sensor data problem. The problem was most severe when a project's scope was changed, such as adding a new module. Nevertheless, specially designed telemetry charts are efficient at detecting the problem. It seems that the best practice would be to designate a person to spend one or two minutes each day to examine the charts for early detection of sensor data problem.

Elaboration:

The central idea of sensor-based metrics collection is that sensors are designed to collect metrics automatically and unobtrusively. Once they are installed, they work silently in the background. It is very easy for a developer to forget about the existence of the sensors. At the same time, it also means that sensor data problem can go unnoticed for a long time. Bad sensor data are caused by many reasons, such as software bug, inappropriate sensor configuration, and server-side project configuration. Though telemetry analysis has greater tolerance for incorrect metrics compared to traditional model-based metrics approaches, complete and correct data still provide the best decision-making value. However, ensuring sensor data correctness is a tedious task. A developer has to log onto the server where raw sensor data are stored and compare the entries with his expectations. It typically involves thousands of sensor data entries on a project of the size like Hackystat.

A serendipitous discovery in this study was that there were several instances that inconsistencies in telemetry charts helped detect the underlying sensor data problem. My hypothesis was that these were not isolated events and that it was possible design telemetry charts to allow the developers to

make quick assessment of the likelihood of the occurrence of sensor data problem. These charts could be used in a number of different ways:

- **Detecting dropout of data points in telemetry streams:**

A dropout usually indicates that the sensor did not send the data. For some types of metrics, it is completely normal. For example, *ActiveTime* is only generated when a developer is actively editing code inside an IDE. It is normal for it drop out for a few days because the developer might be taking a break. But a dropout of *ActiveTime* for a prolonged period of time might be indicative of problem. For other types of metrics, any dropout signifies an error condition. For example, CSDL uses an integration build system to run unit tests and collect coverage metrics every night automatically. There should be no missing data point in coverage data stream if everything is working as expected.

- **Detecting outliers or sudden value changes in telemetry streams:**

An outlier or sudden value change is normal if it is caused by drastic change in software development process or software product. But, often times, it is an indication of sensor breakdown: sending incomplete or incorrect data to the server.

- **Detecting whether related metrics were changing together or not:**

Some related metrics should change together with each other. For example, in Figure 7.18, *active time*, *build*, *unit test*, and *commit* are related because they all serve as proxy for software development effort. If one of them does not co-vary with the rest, it usually indicates that the sensor is not working correctly.

After I made the sensor data verification telemetry charts available, they supported early detection of several instances of sensor data problems. One instance involved Figure 7.17, which showed a chart tracking unit test coverage. The chart was generated for the period from Feb 14 to Feb 26 on a daily interval. There was no unit test data after Feb 20. The sudden drop of coverage from 64% to 55% made it even more suspicious that something significant had occurred to the project around that day which broke the *Emma* sensor.⁶ Further investigation revealed that one of the developers had created a new module, but forgot to update the configuration file to include that module.

Another instance involved Figure 7.18, which showed a chart representing four types of metrics related to software development effort: the number of active time hours, the number of local builds,

⁶ “*Emma*” sensor was the sensor CSDL used to collect unit test coverage information.

the number of unit test invocations, and the number of commits. The four types of metrics should all co-vary with each other, because they all represent software development effort albeit from different perspectives. For example, from Feb 16 to Mar 6, everything was normal. When active time was high, the other three types of metrics were high. When active time was low, the other three types of metrics were low. When active time was zero, the other three types of metrics were zero too. However, for the four days from Mar 7 to Mar 10, the metrics values were abnormal. The developer had build, unit test, and commit activities, but there was no active time. Further investigation revealed that the developer had reinstalled Eclipse IDE, but forgot to reattach the sensor.

The CSDL experience appears to suggest that it is almost impossible to avoid sensor data problem altogether, the best practice would be to designate a person to spend one or two minutes each day to examine the charts for early detection of sensor data problem.

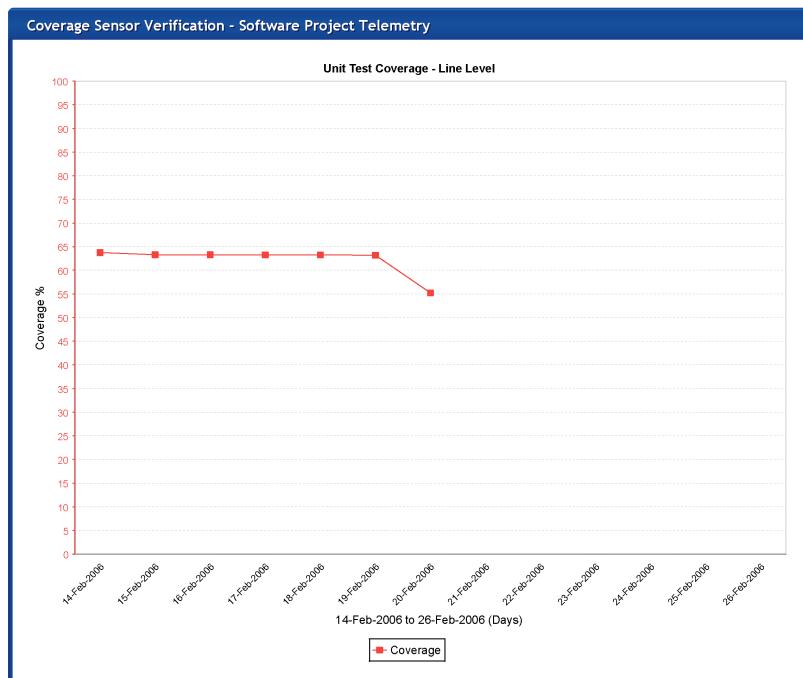


Figure 7.17. Telemetry Chart Indicating “Emma” Sensor not Working

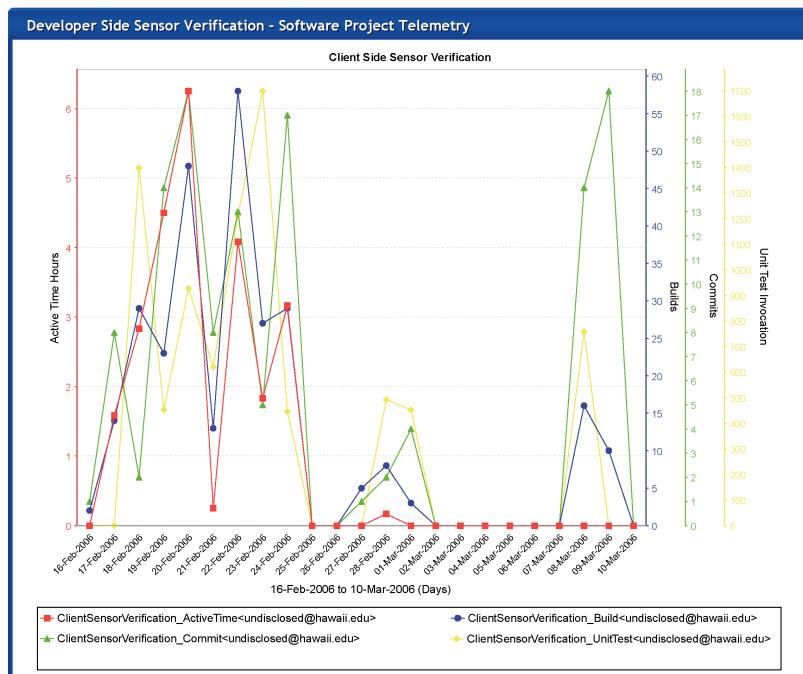


Figure 7.18. Telemetry Chart Indicating “Eclipse” Sensor not Working

7.5.6 Limitation on Low-level Details

Pre-hypothesis Data:

- 2006-02-05-1: I designed telemetry charts for the telemetry wall. Some of the charts were devoted to project level and individual level release cycle issue tracking.
- 2006-02-06-1: I introduced the telemetry wall in the weekly status meeting. While the project level issue tracking charts were found “*highly useful*” by the project manager, the individual level issue tracking charts were completely useless. One developer commented: “*I would rather look into Jira directly, because I don’t know which issues remain.*” The project manager commented: “*I would not be interested in those individual charts.*”

Generated Hypothesis:

The plausible explanation was related to information abstraction. Telemetry analyses offer high level perspectives on software development process by discarding low level details. The project manager’s job was to make sure progress has been made and the software can be released as scheduled. He did not have to know low level details about individual issue. Project level issue tracking charts had the right level of abstraction to help him make release cycle decisions. On the other hand, the developers had to know issue details in order to fix them, but such information had already been discarded by telemetry analyses.

Intervention:

None, except that I removed the individual level issue tracking charts from the telemetry wall.

Post-hypothesis Data:

- 2006-04-25-1: I modified the code issue telemetry chart to track the number of warnings falling into “fail” and “monitor” categories. The chart was primarily designed to be used by the project manager. I enhanced FindBugs report. Warnings in the “fail” category were highlighted in red color, and warnings in the “monitor” category were highlighted in blue color. I also modified the build script so that the developers could generate the report with single

command on their workstations. This was primarily designed to be used by the developers to fix the warnings.

- 2006-05-06-1: Telemetry analysis indicated that all FindBugs warnings in the “fail” category had been eliminated. I had not received any complaint from the developers about the enhanced FindBugs report.

Conclusion:

The post-hypothesis data indicated that I had learned my lesson about information abstraction in telemetry analyses. Apart from the code issue tracking charts, I provided enhanced FindBugs report to supply low level details necessary for the developers to eliminate the potential bugs. Both the issue tracking experience and the FindBugs experience seem to suggest that telemetry analyses will likely to provide decision-making value when a task requires relatively high level information abstraction, but they will not likely to provide value for tasks that require low level details.

Elaboration:

At the early stage of this study, two of the scenes on the telemetry wall were related to release cycle issue tracking. One of them contained charts for project level issue tracking, while the other contained charts for individual level issue tracking. Figure 7.8 and 7.10 were two examples of project level issue tracking charts. The charts were a huge success with the project manager (see Section 7.5.1), because they not only enabled him to track progress within each release cycle, but also established a baseline for future release cycle planning and scheduling. The individual level issue tracking charts were similar to those for project level issue tracking. The only difference was that the individual issue tracking charts were developer-specific, and telemetry streams they contained represented the number of issues assigned to that specific developer instead of all the issues in the project. I provided each developer an individual issue tracking chart, intending to help him manage his assigned issues. However, the developers found the charts useless, and one of them commented: *“I would rather look into Jira (the issue tracking database) directly, because I don’t know which issues remain.”* The project manager found the charts useless too, and he commented: *“I would not be interested in those individual charts.”*

This was very interesting phenomenon. The project level issue tracking charts did not differ too much from the individual level charts. However, the former were found highly useful while the latter completely useless.

My hypothesis was related to information abstraction. Telemetry analyses offer high level perspectives on software development. In other words, the analyses discard low level details. In case of release cycle issue tracking, the project manager's job is to make sure progress has been made and the software can be released as scheduled. He does not have to know about the details of each issue. The project level issue tracking charts offer the right level of abstraction to help him to make project management decisions. On the other hand, the developers' job is to resolve issues. They have to know the details about each issue, which means they have to delve into the issue tracking database to find the information. Besides, the CSDL developers usually have only 3 to 10 open issues to track at any moment. The abstraction in the individual issue tracking charts does not provide any value to them.

As a result, when I helped CSDL improve the utilization of "CodeIssue" metrics and FindBugs reports (see Section 7.5.2), I learned from my previous experience. I used two sets of intervention procedures targeting the project manager and the developers separately. On the manager side, I provided a telemetry chart (see Figure 7.13) showing the number of FindBugs warning in "*fail*" and "*monitor*" categories respectively. The chart was monitored by the project manager as one of the project quality indicators. On the developers' side, I enhanced the original FindBugs report. Warnings in the "*fail*" category were highlighted in red color, and warnings in the "*monitor*" category were highlighted in blue color. For each reported potential bug, the enhanced report not only told the developers the category it belonged to, but also pinpointed the location in the source code from which it was generated.

The FindBugs experience appears to confirm the hypothesis. Both the issue tracking and the FindBugs experiences seem to suggest that telemetry analyses will likely to provide decision-making value when a task requires relatively high level information abstraction, such as project macro-management and high level process analysis. One the other hand, telemetry analyses will not likely to provide value for tasks that require low level details, such as resolving issues and fixing bugs.

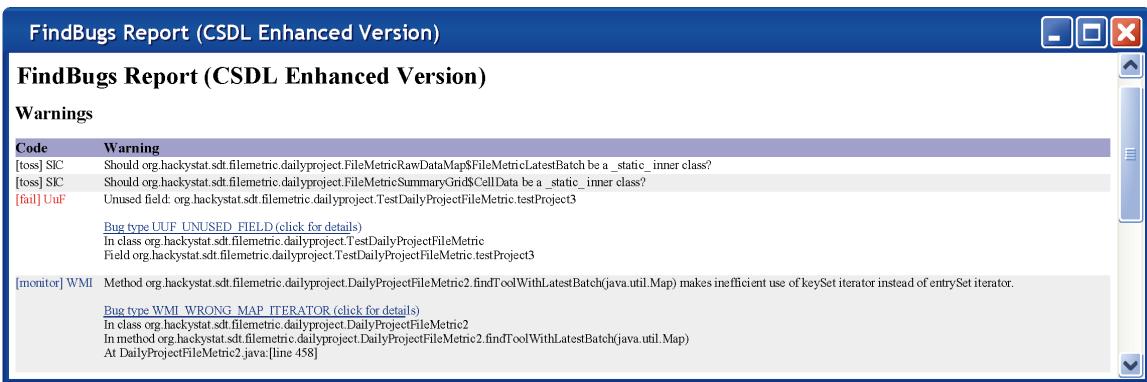


Figure 7.19. CSDL Enhanced Version of FindBugs Report

7.5.7 Telemetry Language Enhancement with Filter Functions

Pre-hypothesis Data:

- 2006-02-20-1: While redesigning telemetry charts for the telemetry wall, I noticed several charts for module-level telemetry trends were overly cluttered (e.g., Figure 7.20).
- 2006-02-21-2: I discussed the charts with three developers. They commented that those charts were too cluttered to be useful. One of them also noted that most of the lines were “uninteresting” because they represented inactive modules.

Generated Hypothesis:

The telemetry language could be enhanced with filter functions to filter out “uninteresting” telemetry streams. This could solve the usability and scalability problem with telemetry charts for large projects.

Intervention:

I augmented the telemetry language to support nested function calls, and implemented special-purpose functions to support filtering telemetry streams in various ways.

Post-hypothesis Data:

- 2006-03-15-5: In an email request for comments, I brought up the idea of introducing nested function calls to the telemetry language.
- 2006-03-16-2: A Jira issue (HACK-612) was created to solve the usability and scalability problem of telemetry charts for large projects.
- 2006-03-17-2: The project manager used the telemetry wall to show the status of Hackystat development to outsider developers. For those overly-cluttered charts, he had to enlarge them to occupy all the nine screens to show the details.
- 2006-04-09-1: I finished the implementation of filter functions and closed the Jira issue (HACK-612).
- 2006-04-11-1: I revised the module-level coverage charts on the telemetry wall by applying filters to show only the top 5 and bottom 5 covered modules (e.g., Figure 7.21). The develop-

ers liked the changes, but they requested a chart to show modules with coverage that changed most.

- 2006-04-17-4: I revised the filter function implantation, and made the chart showing modules with coverage that changed most available on the telemetry wall. One of the developers commented that filter functions made the chart not only much cleaner but also much useful.

Conclusion:

The filter function is a useful enhancement to the telemetry language. It appeared to solve the usability and scalability problem with telemetry charts for large projects.

Elaboration:

The initial telemetry language had only reduction functions, which take sensor data as input and output one or more telemetry streams. In the CSDL study, I noticed a significant usability and scalability challenge with telemetry charts to present “interesting” information from a large number of streams. For a large project, simple telemetry definitions could easily produce charts cluttered with dozens or even hundreds of lines. For example, Hackystat had over 70 modules, one frequent use of software project telemetry was to compare the values and trends of metrics between different modules. One of the charts on the telemetry wall used the following definition to present an overview of module-level test coverage in the Hackystat source:

```
y-axis yAxis(label) = {  
    label, "integer", 0, 100  
};  
  
streams ModuleCoverageStreams() = {  
    "Coverage by Modules",  
    WorkspaceCoverage("Percentage", "***", "line")  
};  
  
chart ModuleCoverageChart() = {  
    "Unit Test Coverage by Modules",  
    (ModuleCoverageStreams(), yAxis("Percent"))  
};  
  
draw ModuleCoverageChart();
```

It generated one telemetry stream for each individual module in the project. The resulting chart was shown in Figure 7.20. It contained over 70 lines, which created a severe usability problem. Discussion with the developers revealed that they generally were only interested in a small subset of those lines, such as the modules with highest or lowest coverage or the modules with coverage that changed most. But it was overwhelming to locate such information in a chart cluttered with over 70 lines.

My hypothesis was that filter functions could be added to the telemetry language to filter out “uninteresting” telemetry streams, which could solve the usability and scalability problem with telemetry charts for large projects. Based on this hypothesis, I augmented the telemetry language to support nested function calls, and implemented special-purpose functions to support filtering telemetry streams in various ways.

The following example illustrated the use of two filter functions:

```
streams ModuleCoverageStreams( ) = {  
    "Coverage by Modules",  
    Filter(  
        FilterZero(  
            WorkspaceCoverage( "Percentage", "##", "line")  
        ), "SimpleDelta", "Bottom", 3  
    )  
};
```

Both “*Filter*” and “*FilterZero*” were the name of filter functions. They were invoked on the stream expression for test coverage for all modules. In other words, the input to the filter functions was the output from the “*WorkspaceCoverage*” reduction function, which was visually represented in Figure 7.20. The “*FilterZero*” function got rid of all lines with only zero values, while the “*Filter*” function further reduced the set of telemetry lines to the three with the most significant decrease in value during the interval. The effect was to produce a chart showing the modules that were decreasing the most in test coverage. The resulting chart was illustrated in Figure 7.21. It contained the modules potentially most in need of additional quality assurance resources.

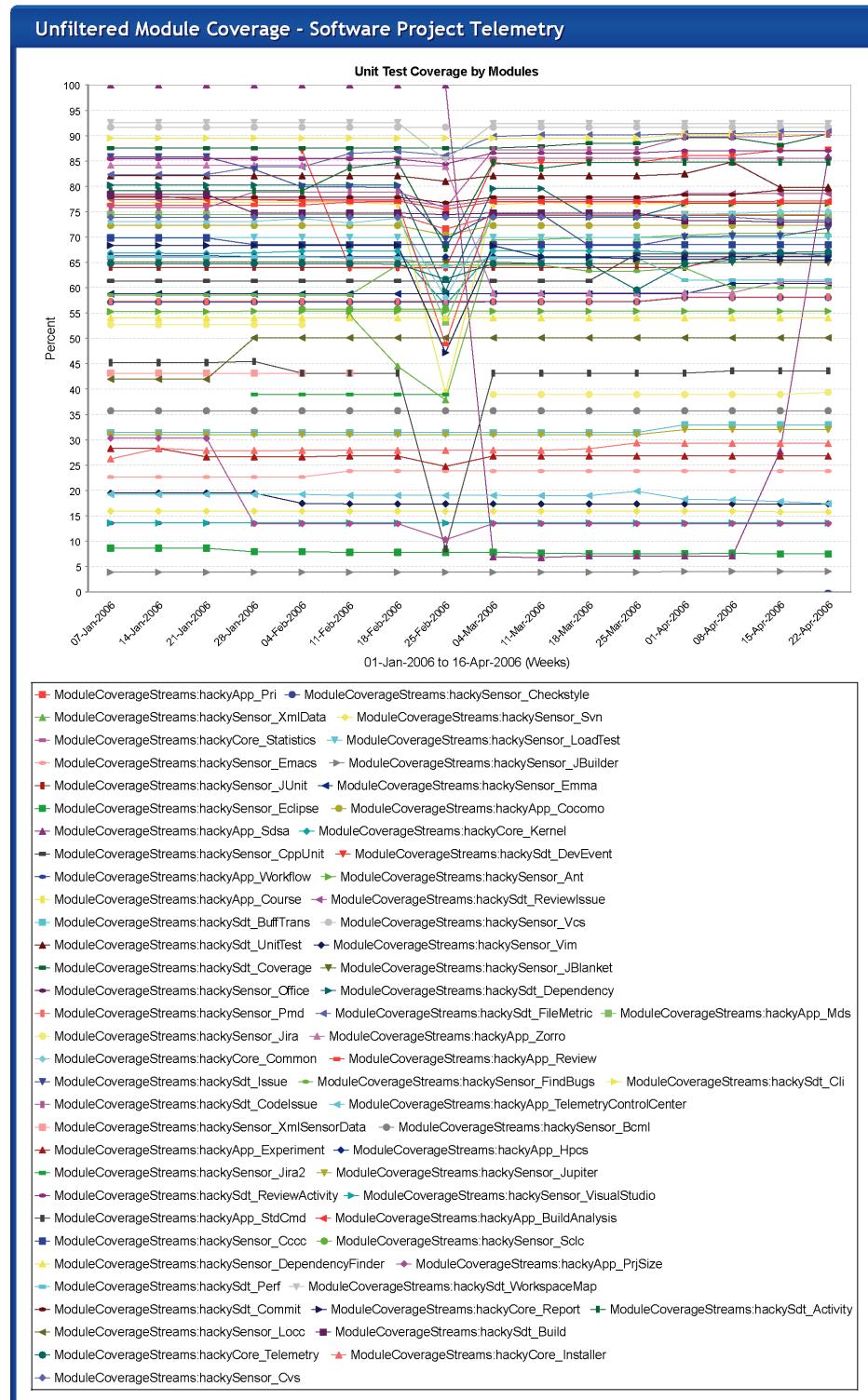


Figure 7.20. Telemetry Chart with Unfiltered Module Coverage

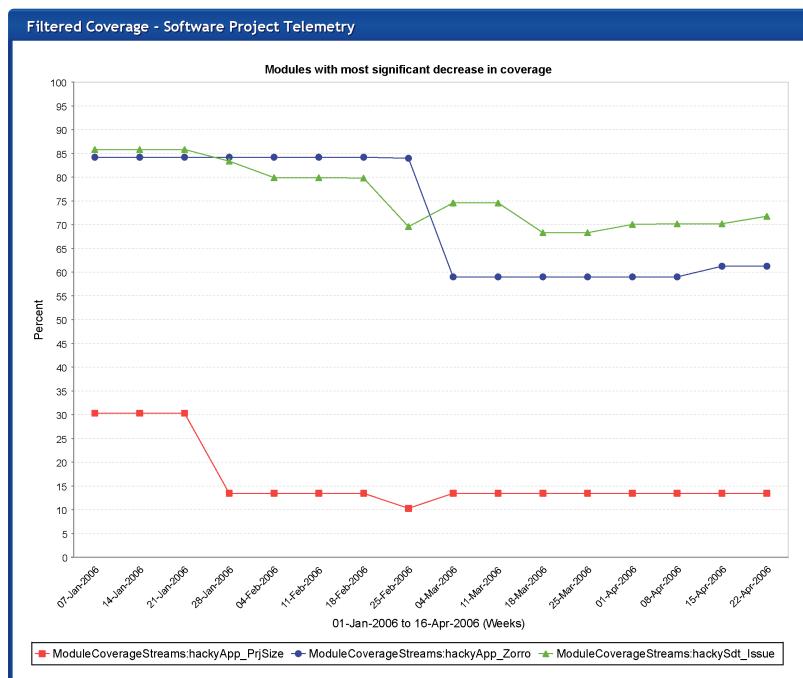


Figure 7.21. Telemetry Chart with Filtered Module Coverage

7.5.8 Telemetry Language Enhancement with Y-axis Construct

Pre-hypothesis Data:

- 2006-02-02-2: I showed some module level coverage charts to a developer. He noted that the automatically-scaled y-axis made comparison across related charts difficult.
- 2006-02-22-1: I discussed the module level quality indicator telemetry scene with the project manager. We had to check the vertical axis while comparing coverage trends in different modules.

Generated Hypothesis:

It would be useful to allow a user to have the option to explicitly specify the range of the vertical axis in a telemetry chart. This way, related charts could be made more comparable by making them have the same vertical axis. The benefit of this extra level of control would outweigh the slight complexity it added to the telemetry language.

Intervention:

I augmented the telemetry language by adding “*y-axis*” construct.

Post-hypothesis Data:

- 2006-03-16-1: During a discussion of telemetry charts with the project manager, I mentioned the idea of making relating charts more comparable by augmenting the telemetry language to allow a user to specify the vertical axis manually. He agreed that it would improve the usability of telemetry charts.
- 2006-03-17-3: A Jira issue (HACK-616) was created to enhance the telemetry language to allow manually specified vertical axis.
- 2006-04-03-1: I held a discussion with the project manager, and we formalized the change to the telemetry language in order to allow a user to specify the vertical axis manually.
- 2006-04-09-2: I enhanced the telemetry language with “*y-axis*” construct, and closed the Jira issue (HACK-616).

- 2006-04-11-1: I update the telemetry wall, converting some charts to use manually-specified y-axes. I showed the changes to the project manager and the developers. They thought the change had improved the usability a lot, because comparisons could be made more intuitively with fixed vertical axes.

Conclusion:

The user feedback after I enhanced the telemetry language with “y-axis” construct suggested that it was a useful feature.

Elaboration:

The telemetry language is designed to be as simple as possible. Most aspects of telemetry presentation are automated, so that a user does not have to fiddle with minute details such as fonts, colors, and layouts. The original language did not have a construct to allow a user to manually specify the range of the vertical axis in a telemetry chart. Instead, it was automatically determined based on the values of the telemetry data points. The result was a simpler language, but, at the same time, different charts might have different value ranges on their vertical axes. This approach worked well in most cases, especially when the range of telemetry data values could not be estimated in advance. However, in the CSDL study, I discovered that it also caused some inconvenience with some charts. For example, Figure 7.22 was generated using definitions written in the original language:

```

streams CoverageStreams(filePattern) = {
    "Coverage",
    JavaCoverage("Percentage", filePattern, "line")
};

chart CoverageChart(filePattern) = {
    "Unit Test Coverage",
    (CoverageStreams(filePattern))
};

```

The two charts showed test coverage for two different modules in the Hackystat source. By cursory examination, you might conclude that coverage in the two modules did not differ too much.

However, if you paid attention to the vertical axes, your conclusion would be very different: the module “*hackyApp_Zorro*” had significantly higher coverage than “*hackyApp_PrjSize*”. Though the utility of the charts was not affected, it was a usability issue nevertheless, and the developers pointed out that the automatically-scaled vertical axes made comparison across charts difficult for such cases.

My hypothesis was that it would be useful to allow a user to have the option to explicitly specify the range of the vertical axis in a telemetry chart so that related charts could be made more comparable, and that the benefit of this extra level of control would outweigh the slight complexity it added to the telemetry language. As a result, I augmented the telemetry language with “*y-axis*” construct, whose syntax takes the following form:

```
y-axis <YAxisName> <ParameterList> = {
    <Label>, <NumberType> (, <LowerBound>, <UpperBound>)?
};
```

‘*LowerBound*’ and “*UpperBound*” are optional fields. A user can choose to provide values to these optional fields to explicitly specify the vertical axis range. Otherwise, the range is determined automatically. With the new “*y-axis*” construct, the telemetry definitions in the previous example have to be updated to a slightly complex form:

```
y-axis yAxis(label) = {
    label, "integer", 0, 100
};

streams CoverageStreams(filePattern) = {
    "Coverage",
    JavaCoverage("Percentage", filePattern, "line")
};

chart CoverageChart(filePattern) = {
    "Unit Test Coverage",
    (CoverageStreams(filePattern), yAxis("Percent"))
};
```

The result was demonstrated in Figure 7.23, which showed the same coverage information for the same two Hackystat modules as in Figure 7.22. But this time the vertical axes in the two charts had the same range from 0 to 100. The difference in coverage for the two modules in Figure 7.23

was obvious, and the comparison could be made more intuitively. The developers welcomed the changes.

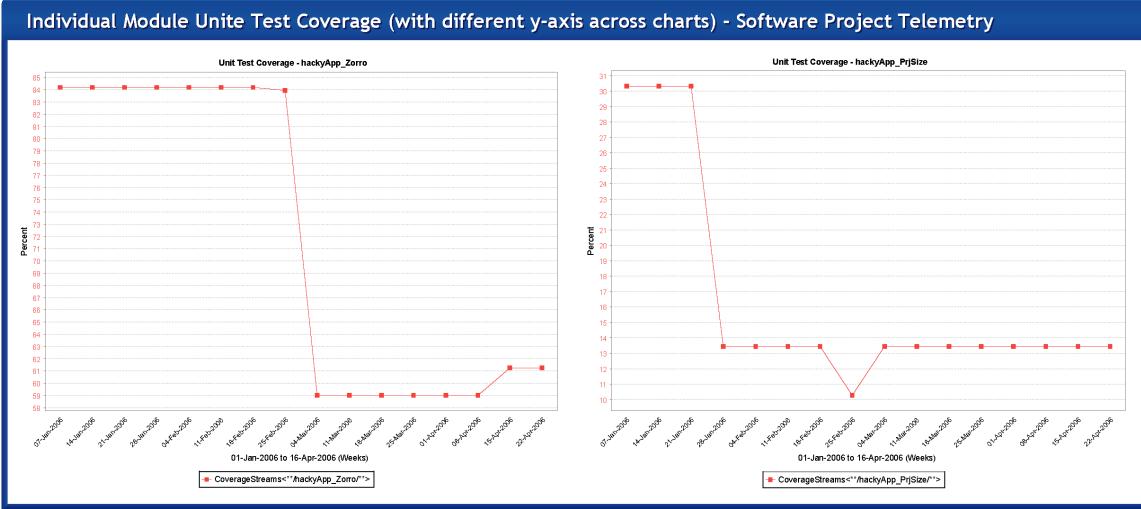


Figure 7.22. Telemetry Charts with Automatically-scaled Vertical Axes

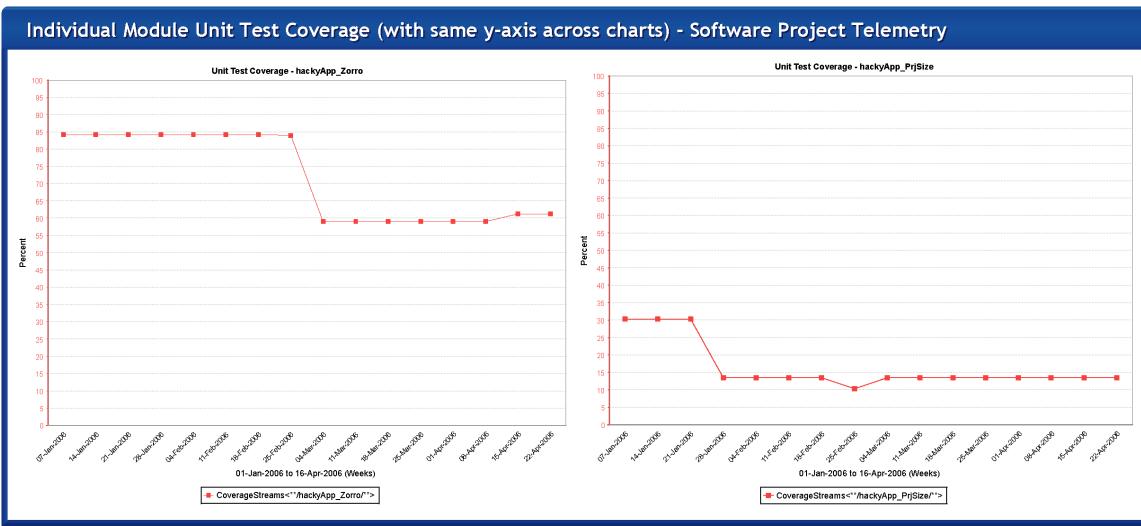


Figure 7.23. Telemetry Charts with Manually-specified Vertical Axes

7.5.9 Runtime Performance Enhancement

Pre-hypothesis Data:

- 2006-02-06-1: I demonstrated the telemetry wall in the weekly status meeting, and noticed that some of the charts covering long-term metrics trends could take several minutes to show up, which was more than the developers were willing to wait.
- 2006-02-13-2: I wanted to display some telemetry charts on the telemetry wall in the middle of the weekly status meeting, but encountered severe server timeout.
- 2006-02-23-1: One of the developers toggled the telemetry wall in auto-update mode. The server stopped responding under the heavy load, and its CPU usage stayed at 100%. I had to restart the server.

Generated Hypothesis:

The performance problem was caused by the bottleneck in the data persistence layer, which used XML files to store sensor data. Given that machine processing of XML data was slow, the solution was to retain as many as possible the most often used sensor data instances in the cache.

Intervention:

I reviewed the code that interfered with the management of the sensor data cache, such as higher level user code not releasing references to sensor data instances. I modified the code to ensure that all references were released promptly.

Post-hypothesis Data:

- 2006-03-01-1: I reviewed the “DailyProjectCoverage” code, and found it that it never released references to sensor data instances, which was temporary data structure as far as “DailyProjectCoverage” is concerned.
- 2006-03-05-2: The project manager created a Jira issue, requesting me to perform an audit on all daily project code to ensure references to temporary data structures were released.
- 2006-03-10-1: I started the code audit, and found several classes followed the same design pattern of “DailyProjectCoverage,” failing to release temporary data structures.

- 2006-03-11-2: After I sent out my findings, the project manager elaborated the distinction between “cache” and “temporary data structure” in an email to call to the attention of all the developers.
- 2006-03-14-1: The server performance improved after the “temporary data structures” were properly disposed, but the improvement was not significant enough to reduce the telemetry wall response time to an acceptable range. I profiled the code with another developer, and we had a number of surprising findings. (1) When sensor data were cached in memory, over 90% of processor time was spent on string comparison in workspace code. The case-sensitive comparison was 10 times more expensive than case-insensitive comparison. (2) The sensor data evolution mechanism in some classes was implemented very inefficiently. For example, the “FileMetric” class spent 80% of processor time in the “recognizeData()” method, even if the data were already stored in the newest format. (3) The cost of reading sensor data from XML files was negligible.
- 2005-03-15-3: I sent out an email to the developer mailing list reporting the performance profiling result.
- 2006-03-15-4: After reporting the performance profile findings, I received an email from a former developer. He had experimented with a Berkeley XML DB back-end and found no performance improvement at all.
- 2006-03-21-2: An external user, who managed his own server, reported degrading telemetry analysis performance proportional to server up-time. He had not picked up the recent fixes, but the problem reported was consistent with the effect of not releasing temporary data structure.
- 2006-04-05-2: There was a performance complaint on the daily project details analysis.
- 2006-04-06-2: In an email, one of the developers noted that there were many redundant computations in the daily project details analysis.
- 2006-04-07-2: One of the developers modified the “DailyProjectUnitTest” code taking advantage of its data access locality, and the result was remarkable: the analysis time for 2006-04-05 unit test metrics was reduced from 124 seconds to 6 seconds.
- 2006-04-10-1: I did the same thing with the “DailyProjectFileMetric” code, and reduced the analysis time for 2006-04-05 file metrics from 180 seconds to just 1 second.
- 2006-04-10-2: The project manager was happy with the results. He wanted to formally document the design pattern as a best practice in the Hackystat developer guide in the summer.

Conclusion:

The generated hypothesis was wrong. The performance problem was not caused by the XML data persistence layer. Instead, the main culprit was the excessive calls to the workspace comparison code.

Elaboration:

The metrics collected by the sensors contain very low level information. For example, a file size metric only captures the size information for one single file. However, telemetry analyses are performed at a much higher project level. They typically have to aggregate hundreds or thousands of metrics for a large project like Hackystat just to compute a telemetry data point representing one single day. Furthermore, analyses performed at the interval of weeks or months are usually produced by first computing a set of analysis at the daily grain size and then combining them together in some fashion to obtain the week or month value. Multiply that by the many different kinds of metrics on which telemetry analyses could be performed, and the fact that the telemetry wall sends multiple concurrent requests in order to display related charts simultaneously, the strain on the server resources could be easily overwhelming.

At the beginning of this study, I found that the telemetry wall response was sluggish at best. It usually took more than several minutes for a telemetry scene to show up. I even experienced several incidents in which requests timed out and I had to restart the server to fix the problem. The sluggish performance caused a usability problem. You just cannot expect a user to push a button and then wait for several minutes for the result in an interactive situation.

Since software project telemetry was implemented as a Hackystat extension, the performance problem was intricately related to the Hackystat framework. The framework stores sensor data in XML files and uses extensive in-memory cache to improve system response time. My hypothesis for the performance problem was that it was caused by the bottleneck in the data persistence layer. Given that machine processing of XML data was slow, the solution was to retain as many as possible the most often used sensor data instances in the cache. Therefore, I started to look for code that interfered with the management of the sensor data cache, such as higher level user code not releasing references to sensor data instances. However, despite my effort of fixing the code, the performance improvement was not significant enough to reduce the telemetry wall response time to an acceptable

range. I began to believe that there was no way to improve the performance except by adding more memory to the server.

The turning point was the day when I profiled the system with another developer. We had a number of surprising findings. We discovered that most processor time was spent on string comparison in workspace code. We also discovered inefficiencies introduced in the recent sensor data type enhancement. However, the most shocking discovery was that the persistence layer was not a bottleneck at all, because the profiler indicated that the time spent reading and parsing sensor data from XML files was negligible. Based on the findings, the code was reorganized to minimize the number of string comparison instead of XML file read. The result was dramatic. In the most significant case, the time computing 2006-04-05 file metrics for all modules in the *Hackystat* project was reduced from 180 seconds to just one second.

My hypothesis was wrong. It reflected the intuition on my part. Though it explained the observed fact in the pre-hypothesis data well, it was inconsistent with the post-hypothesis data. The performance problem was not caused by the XML data persistence layer at all. I learned an important lesson through this experience: application performance improvement should be always guided by profiling data instead of intuition.

7.6 Study Conclusion

The result of this study was positive. It provided direct evidence that software project telemetry has decision-making values. It enabled me to gain a number of valuable insights with respect to the use of software project telemetry. Finally, I was able to improve the telemetry system based on user feedback.

7.6.1 Decision-Making Values

This study provided direct evidence that software project telemetry has decision-making value to the project manager and the developers.

Software project telemetry improved CSDL issue management practice by allowing the project manager to track progress in each release cycle. The historical charts for finished release cycles helped CSDL establish a baseline for future release scheduling and planning. They enabled the project manager to make effort estimation at the beginning of a new release cycle based on the relationship between active time (or calendar days) and the number of resolved issues in previous releases. They also enabled in-process monitoring and control in the middle of a release cycle by comparing the shape of the telemetry from the current release with the shape of the telemetry from previous releases (see Section 7.5.1).

Software project telemetry improved CSDL code quality. By categorizing the warnings reported by *FindBugs* into different severity groups: *fail*, *monitor*, and *toss*, it overcame the false-positive issue inherent in the warnings, gave the developers a better sense of their code quality, and provided them with incentive to eliminate potential software bugs (see Section 7.5.2).

Software project telemetry also provided the developers with insights into their development processes. By providing process feedback, software project telemetry seemed to make the developers more aware of the cost associated with integration build failures, which helped them make more optimal trade-off decisions between reducing local quality assurance cost and reducing integration build failure cost. Though the relationship between software development processes and integration build results were too complex to determine, experience from this study seemed to suggest that with the help of software project telemetry the developers could learn from their past experiences and get “smarter” about their local quality assurance practices (see Section 7.5.3).

7.6.2 Insights

This study enabled me to gain a number of valuable insights with respect to the use of software project telemetry.

Though bottom-up telemetry design based on the types of metrics is able to generate hundreds of telemetry charts without significant effort, the charts lack clear purposes and are generally of little value. You just cannot expect a user to go through hundreds of charts to find the ones that are useful to them. Top-down telemetry design based on user goals, which is similar to the methods in the Goal-Question-Metric paradigm, yields most useful telemetry charts (see Section 7.5.4).

In contrast to the principle behind top-down telemetry design, it is best to collect whatever metrics you can regardless whether you need them or not. The rationale is very simple. Software project telemetry uses sensors to collect metrics automatically and unobtrusively. Given the low cost of sensor-based metrics collection, it could only benefit an organization to collect the extra metrics. Even though you don't need a metric today, it can still be used to establish a baseline for comparison tomorrow. For example, “*CodeIssue*” metrics were considered useless by most developers at the beginning of this study, but it was later used to improve CSDL code quality (see Section 7.5.2).

It seems that sensor data problems are unavoidable. Most problems I encountered during this study were caused by failures to update corresponding configuration when there is a change in the project, such as adding a new module. The good news is that software project telemetry can be used for self-validation. By monitoring data point dropout or sudden value change in telemetry streams, or violation of correlation rules in related metrics, a project manager or a developer can make quick assessment whether there is underlying sensor data problem or not. Therefore, it is highly recommended for a development team to set up telemetry charts specially designed for the purpose of sensor data verification, and appoint a dedicated person to examine these charts for early detection of sensor problem (see Section 7.5.5).

This study also revealed a limitation with software project telemetry. Telemetry analyses are designed to offer high level perspectives on software development process by discarding low level details. The experiences with “*Jira*” issue tracking and “*FindBugs*” metrics seem to suggest that telemetry analyses will likely to provide decision-making value when a task requires relatively high level information abstraction, such as project macro-management and high level process analysis.

One the other hand, telemetry analyses will not likely to provide value for tasks that require low level details, such as resolving issues and fixing bugs (see Section 7.5.6).

7.6.3 Telemetry System Improvement

Finally, I received valuable feedback from the developers about telemetry chart presentation. Based on the feedback, I was able to enhanced the telemetry language to improve the usability of telemetry charts (see Section 7.5.7 and 7.5.8). This study also helped me understand runtime performance characteristics of telemetry analysis for large projects like Hackystat. After optimizing my code, I was able to improve analysis response time several-fold (see Section 7.5.9). All these improvements contributes to a smoother user experience with software project telemetry.

Chapter 8

Evaluation Conclusions

The previous two chapters reported on the evaluation results from the classroom and CSDL studies respectively. This chapter synthesizes the results to gain further insights. Section 8.1 reports the insights. Section 8.2 describes possible future directions of continued software project telemetry evaluation in the next stage on its evolution path.

8.1 Synthesis of Study Results

The evaluation of software project telemetry was carried out in two studies: one in the classroom, and the other in CSDL. They were all mixed-methods studies. The class room study was relatively simple. It was a one-shot survey based study. I distributed a questionnaire at the end of the study to collect the students' opinions about software project telemetry, and their telemetry analysis invocation pattern was analyzed to cross-validate their opinions with the actual system usage. The CSDL study was much more comprehensive. I pursued an in-depth data collection and analysis strategy over a much longer period of time: I gathered data through observations and interviews; I generated hypotheses from the data; I also tested the hypotheses in a limited way.

The reason why different methods were used in the two studies was because the two environments have very different characteristics. The classroom study involved a relatively large number of participants (25 students) working on small-scale class projects. On the other hand, there were a relatively small number of participants in the CSDL study: five developers plus one project manager. However, the software under development was much larger in scale. It contained almost 300,000 lines of code in total, and had been under development for five years. The developers had signifi-

cantly more software engineering experience and process maturity compared to the average student in the classroom.

In both the classroom and CSDL studies, my focus was on the understanding of the use of software project telemetry within the particular software development environments in which the technology was deployed. However, by comparing and contrasting the results from the two studies, I was able to gain a number of insights that might generalize to other software development environments . My insights are grouped into five categories:

- Metrics Collection
- Analysis Invocation
- Decision Making
- Best Practice
- Adoption Barrier

8.1.1 Metrics Collection

Software project telemetry uses sensors to collect metrics. Sensors must be installed and configured. Once they are installed and configured properly, they collect metrics automatically and unobtrusively. The classroom study indicated that the sensor-based metrics collection had achieved the design goal of eliminating the long-term chronic “*context-switch*” overhead inherent in manual and tool-assisted approaches. However, it identified one weak link in sensor-based approach: most students complained that the installation and configuration of sensors were overly complex, and one of them even reported trouble at the end of the semester.

Based on this feedback, an installer was developed in Fall 2005. The installer provides an intuitive graphical user interface that helps a user to download, configure sensors, and check for updates. The installer was used in the CSDL study. The result indicated that it saved the developers a considerable amount of effort installing and configuring the sensors.

However, the CSDL study revealed another weak link in sensor-based metrics collection. Sensors are designed to work silently and unobtrusively in the background. Since metrics collection does not require developer intervention, it is very easy to forget about their existence. As a result, broken sensor data might go undetected for a long period of time. Though software error can cause broken sensor data, the most common cause in the CSDL study was the change in the scope of a

project. For example, in one incident, a developer created a new module but forgot to update the configuration file to include that module, causing a loss of coverage metrics for almost one week. Change in development tools can also cause broken sensor data. For example, in another incident, a developer upgraded his IDE on his workstation to a newer version but forgot to run the installer to reattach the sensor, causing a loss of software development activity metrics for at least two days. The experience seemed suggest that it would be very hard to avoid broken sensor data completely. An interesting observation was that the project manager and the developers were using telemetry charts to help them make quick assessment whether the underlying sensor data are being collected correctly or not. They used a set of heuristic rules to flag suspicious metrics data, such as dropout of data points, outliers or sudden value changes in telemetry streams, and related metrics not changing together.

In summary, the experiences from the two empirical studies indicated that sensor-based metrics collection eliminated the long-term chronic “*context-switch*” overhead, and that the installer significantly reduced the cost associated with the installation and configuration of these sensors. However, broken sensor data are a threat one cannot ignore, especially when the scope of a project is changed. The automated nature of metrics collection might even prevent early detection of this problem. The good news is there are telemetry charts and heuristic rules to help make quick assessment whether sensor data are likely broken by flagging suspicious metrics. But a real person familiar with the project and the development activities is required to make the final judgment. Therefore, it is highly recommended for a development team to set up telemetry charts for the purpose of broken sensor data detection, so that the project manager or a process expert could spend one or two minutes each day examining these charts to make a quick assessment.

8.1.2 Analysis Invocation

The software project telemetry implementation exposes three analysis interfaces: *telemetry chart analysis*, *telemetry report analysis*, and *telemetry expert analysis* (Section 4.2.1). While the telemetry expert analysis requires a user to use the telemetry language to interact with the system, the telemetry chart and report analyses allow an expert to predefine telemetry definitions so that a regular user can perform analysis by selecting from a set of predefined definitions.

In the classroom study, the telemetry language was not introduced. Instead, the instructor and I predefined a set of charts and reports that we thought were most useful to the students, and only

the telemetry chart and report analyses were introduced. To make up for the loss of flexibility of not using the telemetry language, the telemetry chart and report definitions were parameterized. For example, we defined a telemetry chart showing how unit test coverage varies over time, which used a parameter called “*FilePattern*”. The idea is to allow the students to specify which files should be included in the computation. For example, while a parameter value “`**`” computes coverage for the entire project; “`foo/**`” computes coverage for only the *foo* module in the project. The parameterization solved a practical problem of combinatorial explosion of telemetry definitions, because telemetry inquiries often led to the wish to compare the same telemetry streams across different developers, different modules, or even some combination of developers and modules. The survey responses indicated that the students appreciated the idea of being able to run metrics analysis by choosing from a list of predefined charts and reports, and most of them agreed that it was simple to invoke the analyses.

In the CSDL study, the telemetry wall was used to communicate telemetry analysis results to the developers. I took full responsibility of gathering metrics analysis requirements, designing telemetry charts, and making them available on the telemetry wall. Though the CSDL developers were not expected to run telemetry analysis themselves, I asked them their opinions once during an interview about the three analysis interfaces provided by the software project telemetry implementation. One developer told me that “*the (telemetry) language is the last thing I want to use*” because it looks complex.

The complexity of the telemetry language is necessary, because different software development environments have different constraints for metrics collection and different requirements for metrics analysis. A metric available in one environment might not be available in another environment; and a useful analysis in one environment might not be that useful in another environment. The language provides a glue mechanism so that we can decouple the types of metrics we are able to collect and the types of analysis we wish to support. Despite the fact that the classroom study and the CSDL study were all conducted in academic settings, different types of metrics were collected, and different sets telemetry charts/reports were used in the two environments.

The experiences from the two empirical studies confirmed that an effective approach to this language complexity problem is to hide the language from regular users. A power user or a process expert can take the responsibility of designing telemetry charts. For example, in the classroom study, the instructor and I predefined charts and reports for the students. In the CSDL study, I took the sole responsibility of designing charts and making them available on the telemetry wall.

However, the classroom study did reveal one usability problem with the telemetry chart and report analyses. The problem was related to the input of parameter values. Different charts or reports had different parameter requirements, but it was hard to tell from the simple web interface what parameters were expected. The current solution is: (1) continue to use parameters in telemetry chart analysis but provide detailed error messages, so that the user can find out the expected parameters through trial-and-error; (2) establish a convention that all telemetry reports are defined without parameter. The solution represents a compromise, because software project telemetry is implemented as a Hackystat extension, and it has to abide by its user interface constraints. The Hackystat UI constraints are largely vestiges from the old days to support stone-aged web browsers running on antediluvian Unix platforms. Fortunately, in the last two years, the web browser market has consolidated into only two major rivals (i.e., Internet Explorer and FireFox), and sophisticated ajax-based¹ rich Web UI frameworks such as GWT [81] and Echo2 [25] have emerged. The Hackystat developers are discussing the possibility of migrating toward such a framework. Once the migration is complete, the user interface of the telemetry chart and report analyses can be redesigned to provide real time hints about the types of parameters expected.

In summary, the experiences from the two empirical studies suggested that the complexity of the telemetry language is necessary to meet real world metrics collection constraints and analysis requirements. However, the complexity does not have to be exposed to a regular user of software project telemetry. A process expert or a power user can predefine telemetry charts and reports for them. In fact, a very important step in using software project telemetry is the customization of telemetry charts and reports to the specific need of a software organization.

8.1.3 Decision Making

Collecting metrics is only a means; the end goal is to make decisions. The idea is that by performing telemetry analyses over collected metrics, developers gain insights into their software development processes, which, in turn, help them make process improvement decisions. One of the objectives of the classroom and CSDL studies was to assess the value of software project telemetry for decision-making.

¹Ajax is an acronym for “asynchronous JavaScript and XML.” It is a technique for creating highly interactive web applications, by utilizing client-side JavaScript and the DOM model to update a portion of a web page according to the result of an asynchronous server method invocation.

The classroom study was designed as a “passive” study. The students were left on their own to experiment with software project telemetry to gain insights from their process and product metrics. The survey responses showed strong evidence that software project telemetry made their development processes transparent. In fact, some students even seemed to suggest that it made their processes more transparent than they had wished by raising privacy concerns. However, the evidence was inconclusive whether software project telemetry had helped them improve their processes.

In the CSDL study, I took an “active” role as an on-site process expert by introducing software project telemetry as a process improvement program. I followed the steps of analyzing the lab’s process status, customizing telemetry analyses, proposing improvement recommendations, and finally validating process changes. These steps resulted in conclusive evidence that software project telemetry had decision-making values: I was able to help not only the project manager institute changes to improve project management practices, but also the developers gain insights into their software development processes.

Several plausible reasons could explain the different results in the two empirical studies.

First, the experiences seem to suggest that the ability to understand and interpret telemetry data might have a strong impact on telemetry decision-making values. There were several reports during the classroom study in which the sensors did not seem to collect data correctly, or the telemetry analyses did not seem to compute the expected results. It turned out that some of them were caused by either misunderstanding of metrics or inappropriate interpretation of the analysis results. On the other hand, there were no such problem in the CSDL study.

Second, the experiences also seem to suggest that process maturity might affect telemetry decision-making values. The students in the classroom had relatively low process maturity on average because they were still learning software application lifecycle techniques, but the CSDL developers had much higher process maturity levels. Higher process maturity leads to greater visibility into software development practices. As a result, more metrics could be collected and analyzed in CSDL, resulting in more opportunities for process improvement. Indeed, I was able to collect metrics about project issue tracking, code review, and integration build in the CSDL study, but not in the classroom study.

Lastly, it seems that the ability to customize telemetry analyses to the specific need of a project might also have an impact on telemetry decision-making values. In the classroom study, though the instructor and I predefined some telemetry charts and reports for the students based on what

we thought were most useful to them, we did not update these charts and reports with feedback from them. On the contrary, in the CSDL study, I not only customized telemetry analyses but also validated their usefulness according to my observational data and developer feedback.

8.1.4 Best Practice

“*Top-down telemetry design*” and “*bottom-up metrics collection*” seem to be best practices in software project telemetry. Top-down telemetry design refers to the idea that each telemetry chart should have a clear purpose, such as to help the development team meet a specific process improvement goal. This is similar to the idea in the Goal-Question-Metric paradigm. Bottom-up metrics collection refers to the recommendation to collect whatever metrics a software organization can. Software project telemetry does not suffer from the metrics collection cost problem because sensor-based approach eliminates the chronic “context-switch” overhead inherent in manual and tool-assisted approaches. Even if there is no apparent need for a metric today, it can still be used to establish a baseline for tomorrow.

8.1.5 Adoption Barrier

One adoption barrier involves data privacy concerns. This issue seems most severe with effort-related individual developer process metrics, such as “*ActiveTime*”. An interesting observation is that this issue only manifested itself in the classroom study. There was no indication of data privacy concern in the CSDL study. The plausible reasons are: (1) Some students misunderstood “active time” for “software development effort,” while there was no such misunderstanding among CSDL developers; (2) The classroom group projects emphasized on member participation and *active time* was the primary measure, while progress in CSDL was measured by “*Jira*” issue closure instead of *active time*. In fact, the data privacy issue has been identified in many literatures. For example, Grady [35] suggests that specific rules be implemented regarding who can access what portion of data, and when data go from private to public. The recommendation is listed in Table 8.1. The software project telemetry implementation has a mechanism to limit the scope that the metrics can be accessed. But overcoming this data privacy issue seems largely dependent on what the data are used for in an organization. In other words, are the data used for process improvement or developer performance evaluation?

Table 8.1. Data Access Recommendations by Grady

Individual	Project Team	Organization
Defect rates (by individual)	Defect rates (team)	Defect rates (by project)
Defect rates (by module)	Module size	Size (by product)
Defect rates (under development)	Estimated module size	Effort (by project)
Number of compiles	Number of re-inspections	Calendar times
	Defects per module (pre-release)	Defects per module (post release)
		Effort per defect (average)

Another adoption barrier involves telemetry expertise. The experiences from the empirical studies indicate that software project telemetry will not likely to delivery best value when used “out-of-the-box.” Different projects have different goals and different requirements. To get the best value out of software project telemetry, it needs to be customized. The customization includes not only setting up sensors to collect metrics, but also designing telemetry charts for the specific needs of the project. For example, in the CSDL study, I acted as an on-site process expert. I took the iterative steps of analyzing current software processes, identifying improvement opportunities, designing telemetry charts, implementing changes, and evaluating the results. The experience seems to suggest that software project telemetry can provide decision-making values when this iterative approach is followed. However, there might be difficulty for an organization adopting the technology to find somebody who can perform a similar role to the one I performed in the CSDL study.

8.2 Future Evaluations

The classroom and CSDL studies enabled me to get a basic level of understanding with respect to the real world use of software project telemetry. However, to various degrees, my knowledge was tied to the academic environments in which the technology was deployed. Though the CSDL setting shared many characteristics with industrial settings, such as large project size and experienced developers, there were still some important elements missing from a typical commercial environment, such as project deadlines and budget constraints. Therefore, it is desirable to evaluate software project telemetry in different industrial settings. Will it be possible to arrive at the same conclusions about software project telemetry in industrial settings? If not, what would be the differences between the software development environments that could account for different conclu-

sions? A constructivist approach could give in-depth understanding about the “context” to answer these questions.

An interesting observation from the classroom and CSDL studies is that process maturity seems to have a strong impact on decision-making value of software project telemetry. Since higher process maturity offers greater visibility into software development activities, more metrics can be collected and analyzed. A future area to explore would be how software organizations with different maturity levels use software project telemetry. Does successful use of software project telemetry exhibit different patterns in a low process maturity organization than in a high process maturity organization?

Another area to explore is the impact of organizational structure on telemetry data collection. Some organizations are more centralized with collocated developers, while others are more decentralized with geologically-dispersed developers collaborating with each other. Different organizations have different constraints on metrics collection. The current experiences suggest that one significant adoption barrier to software project telemetry is the issue of data privacy and confidentiality. This concern is most severe with personal process metrics, especially effort-related ones. On the other hand, most product metrics and certain process metrics do not seem to cause such a problem, such as information about file size, unit test coverage, source code commit, and bug/issue status. Two interesting questions are: to what extent will the loss of personal process metrics affect the decision making value of software project telemetry? What can be done to alleviate the privacy concern in an industrial environment?

Chapter 9

Final Remarks

This chapter presents some final remarks about this thesis research. It begins with a summary of the research in Section 9.1. It then lists main contributions in Section 9.2. Finally, it discusses future directions in Section 9.3.

9.1 Research Summary

This research was inspired by many existing software measurement approaches, such as PSP [42, 43], COCOMO [10, 11], and CMMI [73]. These approaches aim to systematically improve software development processes in order to enable a software organization to produce software products in a controllable and repeatable manner. But the “*metrics collection cost problem*” and “*metrics decision-making problem*” make effective application of these approaches far from mainstream in practice. The “*metrics collection problem*” refers to the high cost associated with metrics collection; while the “*metrics decision-making problem*” refers to the question how to make project management and process improvement decisions based on the information in the metrics. They are the motivation behind this thesis research.

My solution is a novel approach to software measurement called “software project telemetry,” which enables (1) automated metrics collection and analysis, and (2) in-process, empirically-guided software development process problem detection and diagnosis. Software project telemetry addresses the “*metrics collection cost problem*” through highly automated measurement machinery: software sensors are written to collect metrics automatically and unobtrusively. Sensors keep metrics collection cost low by eliminating the chronic “context-switch” overhead inherent in both man-

ual approaches, such as PSP, and tool-assisted approaches, such as LEAP [65]. Software project telemetry addresses the “metrics decision-making problem” through a domain-specific language designed for the representation of telemetry trends for different aspects of software development process. Project management and process improvement decisions are made by detecting changes in telemetry trends, and comparing trends in two different periods of the same project, instead of between two completed projects found in model-based approaches such as COCOMO. The advantage is that it not only eliminates the need to build statistical models that require frequent calibration, but also enables in-process control for a project that is still being developed.

In order to evaluate software project telemetry, I conducted two empirical studies: one in the classroom, and the other in CSDL. The research followed, for the most part, the constructivist paradigm. The classroom study was relatively simple. The students were divided into groups of 2 - 4 members collaborating on group projects. They were introduced software project telemetry to collect metrics and perform analyses on their own data. At the end of the study, I distributed a questionnaire to collect the students’ opinions about software project telemetry. I also analyzed their telemetry analysis invocation pattern to assess the extent to which their opinions were based on the actual system usage. On the other hand, the CSDL study was much more comprehensive. I introduced software project telemetry as a metrics-based process improvement program. I pursued an in-depth data collection and analysis strategy over a much longer period of time. It involved many methods from the constructivist paradigm: it was a case study, in which I collected data through observations and interviews and generated hypotheses from the data. It also involved a limited form of hypothesis testing in the post-positivist tradition.

The results from the two studies suggested that software project telemetry had acceptably-low metrics collection and analysis cost, and that it was able to provide project management and process improvement decision-making values. They also suggested that software project telemetry would deliver best value when metrics collection and telemetry analysis could be customized to the specific need of a software organization. Top-down telemetry design and bottom-up metrics collection are best practices. However, data privacy concerns among developers and lack of telemetry expertise in organizations outside CSDL might become technology adoption barriers.

9.2 Dissertation Contribution

There are three main contributions from this research:

1. the idea of software project telemetry itself,
2. the system implementation,
3. and, finally, the insights gained from the two studies.

9.2.1 Concept of Software Project Telemetry

First, the idea of software project telemetry itself is a significant contribution. It uses sensors to collect metrics, which represents a leap from traditional manual and tool based approaches. These traditional approaches require human intervention or developer effort to collect metrics. The developers are constantly distracted from their main work (i.e., developing software), because they have to switch back and forth between doing the work and recording what work is being done. Several studies [12, 42, 53] have identified the high metrics collect cost as a major adoption problem. On the contrary, in a sensor-based approach, sensors collect metrics automatically and unobtrusively. It significantly lowers the metrics collection cost by eliminating the chronic “context-switch” overhead inherent in manual and tool based approaches. It also has an added advantage: metrics collection no longer suffers from bias (either deliberate or unconscious), error, omission, or delay, and we don’t have to worry about the types of data quality problems reported in [52].

In metrics decision-making, software project telemetry follows a light-weight approach. It represents a deviation from traditional model-based approaches such as COCOMO. Since software project telemetry does not compare the current project with the projects in historical database, there is no need to build formal models based upon statistics over previously completed projects. This avoids many problems inherent in traditional metrics models, such as the need to accumulate a historical project database and ensure that the historical data remain comparable to current and future projects. Instead, software project telemetry makes comparisons between two different periods of the same project, and the comparison involves much smaller time scale than the whole lifecycle of a project. The metrics from the initial period of the project are used to establish a baseline and bootstrap the process. Project management and process improvement decisions are made by detecting changes in telemetry trends and comparing trends in two periods of the same project. In-process

control for a project that is still being developed is made possible exactly because comparisons are made within the same project. It also solves a dilemma for many software organizations with low process maturity: these organizations simply have no basis to perform model-based cross-project comparison because their software development processes change as work changes.

The classroom and CSDL studies showed that software project telemetry had sufficiently low metrics collection and analysis cost, and that it was able to deliver project management and process improvement decision-making values at least within the exploratory context of the two studies. In the classroom study, most students responded that they felt their software development processes had improved. Though I was unable to ascertain the degree to which the students' self-claimed improvement was caused by the use of software project telemetry, it was certain that software project telemetry made their software development processes transparent. In the CSDL study, by introducing software project telemetry as a metrics-based process improvement program, I was able to help the project manager institute changes to improve project management practices, and help the developers gain insights into their software development processes.

9.2.2 Implementation of Software Project Telemetry

Second, the implementation of software project telemetry is also a significant contribution. Two pieces of software are the direct result of this thesis research. One of them is a server-side component of about 28,000 lines of code (the Core_Telemetry module plus the custom implementation of telemetry reducers and functions). It includes the software code to interpret the telemetry language, and the code to perform telemetry analyses and generate telemetry charts. It also includes a web-based management console for telemetry construct definitions. A user can perform three types of analysis using a web browser: the expert analysis, the chart analysis, and the report analysis. The other piece of software is the telemetry control center (Figure 7.3) of about 3,000 lines of code. It is a client-side application that can be configured to automatically retrieve telemetry charts from the server and display them. Though it was deployed on the telemetry wall during the CSDL study, it can also run on a standard personal computer.

The two pieces of software enable two complementary modes of operation. In the first case, a user logs on to the server using a web browser to “*actively*” explore relationships between different software metrics. In the second case, the telemetry control center makes a sequence of telemetry charts continuously available to the user, providing “*passive*” awareness of the project status.

The performance of the server component has been profiled and fine-tuned. It is capable of handling a large amount of sensor data generated by enterprise-level large-scale software projects and multiple concurrent requests smoothly. For example, it is being used by CSDL to analyze Hackystat product and process metrics, and Hackystat had nearly 300,000 lines of code in total.

The source code is GPL licensed. It is freely available. The GPL license ensures that any third-party improvement will be contributed back to the community. An added advantage is that the code has become a standard component of the much wider scoped Hackystat project, which means it will be actively maintained for a long time. The system has already been adopted by several external sites, such as Sun Microsystem and the University of Maryland.

9.2.3 Insights from Empirical Studies

Lastly, there are a number of valuable insights by comparing and contrasting the results from the classroom and CSDL studies. I grouped my insights into five categories:

- **Metrics Collection** — Sensor-based metrics collection appears to have eliminated the long term “context-switch” overhead inherent in manual-based approaches, such as PSP, and tool-based approaches, such as LEAP, PSP Studio, and Software Process Dashboard. The installer appears to have reduced the one-time sensor setup cost considerably. However, due to the fact that sensor collects metrics automatically in the background, broken sensor data might go unnoticed for a long time. The good news is that it is possible to design special-purpose telemetry charts to help developers make quick assessment whether the underlying sensor data are likely broken or not.
- **Analysis Invocation** — The telemetry language appears to be quite complex for a normal user. It seems a good idea to have a telemetry expert pre-define telemetry charts and reports before-hand, so that a normal user could select from a list of pre-defined definitions to invoke the analysis. This is exactly the idea behind the telemetry chart/report analyses in the current implementation. However, there was a usability issue with respect to the input of parameter values. Fortunately, the issue appears resolvable by switching to a richer Web UI framework.
- **Decision Making** — There is clear evidence that telemetry analysis makes software development process transparent. However, whether a team or a developer can get decision-making values out of it seems to depend on a number of factors, which include the level of process

maturity, the ability to understand and interpret telemetry data, and the ability to customize telemetry analysis to the specific need of an organization.

- **Best Practice** — “Top-down telemetry design” and “bottom-up metrics collection” appears to be best practices of software project telemetry. Top-down telemetry design refers to the idea that each telemetry chart should have a clear purpose, such as to help the development team meet a specific process improvement goal. Bottom-up metrics collection refers to the recommendation to collect whatever metrics a software organization can.
- **Adoption Barrier** — One technology adoption barrier involves data privacy concerns, which seem most severe with effort-related personal process metrics, such as “*ActiveTime*”. Though the current implement of software project telemetry has a mechanism to limit access to metrics, overcoming this issue largely depends on what the metrics are used for in an organization (i.e., process improvement vs. performance evaluation). Lack of telemetry expertise within an organization might be another technology adoption barrier. Software project telemetry will not likely deliver the best value if used straight “out of the box.” Effective use requires customization, which includes both setting up the sensors to collect metrics and designing telemetry charts and reports.

9.3 Future Directions

Software engineering is highly contextual. Two recurring questions in the classroom and CSDL study were: what is good telemetry, and how do we recognize bad trends? The answers are most likely dependent on the type and the goal of the project. Currently, software project telemetry requires human judgment to make project management and process improvement decisions. Is it possible to provide some degree of automated decision-making support? To what extent can such support be automated? Data mining provides an interesting direction for future research.

Data mining is the process of extracting valid, authentic, and actionable information from large databases. In other words, it derives patterns and trends that exist in data. There are many commercial softwares that offer sophisticated data mining support. Most of them are based on OLAP (On Line Analytical Processing). The most fundamental data structure in OLAP is multi-dimensional cube. A cube is a set of measures, which are facts, and dimensions, which are areas of interest. Measures are data, and dimensions define the ways data can be summarized. A dimension can have

multiple hierarchies. The analogy is Google map: when viewed from a distance, you see the entire city or the entire country; when zoomed in, you see houses and streets.

There is a high degree of similarity between Hackystat and OLAP: a cube in OLAP corresponds to a project in Hackystat. Measures are equivalent to sensor data (a.k.a. software metrics); dimensions are members, workspaces, and time intervals for a project. The difference is that a cube operates at a higher level of abstraction than a Hackystat project: it treats members, workspaces, and time intervals in a uniform way. This similarity makes it easy to take advantage of wide range of data mining algorithms in existing commercial software packages. For example, Microsoft SQL Server Analysis Services 2005 (SSAS) has several classes of data mining algorithms:

- **Classification Algorithms** — They predict one or more discrete variables, based on the other attributes in the dataset.
- **Regression Algorithms** — They predict one or more continuous variables, based on the other attributes in the dataset.
- **Segmentation Algorithms** — They divide data into groups, or clusters, of items that have similar properties.
- **Association Algorithms** — They find correlations between different attributes in a dataset.
- **Sequence Analysis Algorithms** — They summarize frequent sequences or episodes in data.

The introduction of data mining to software project telemetry could open some very interesting new possibilities for metrics analysis. In the past, the focus of software project telemetry analysis has been on *manually* detecting covariance between software process metrics and product metrics. The idea is that if we can identify the relationship between them, then we can increase product quality by controlling the process. This seems to be a special case of an association algorithm: identifying correlations between different attributes in a dataset. Therefore, one possibility with data mining would be to investigate the extent to which the manual process of identifying covariance in telemetry streams could be automated by using association algorithms.

Another possibility would be to apply sequence analysis algorithms to recognize software engineering best practices. For example, we may wish to identify process sequences in telemetry streams that correspond to decreasing number of reported bugs in a project. Sequence analysis algorithms

find the most common sequences by grouping identical sequences together. They have long been used by online retailers to recommend related purchases. A related research currently supported by Hackystat is SDSA [57], which analyzes software process metrics to classify them as indicating the use of a best practice. The difference is that SDSA requires *a priori* knowledge of what constitutes software development best practices, but sequence analysis algorithms have no such requirement.

Yet another possibility would be to apply regression algorithms for statistical process control. Existing telemetry streams establish baseline values for various software development measures in a software organization. Assuming the process is stable, then regression algorithms could be used to find statistical control bounds to send alert if new metrics values fall outside the bounds. The idea is similar to six sigma, which is a project management methodology that uses data and statistical analysis to measure and improve a company's operational performance.

An important note is that using of data mining approaches to automate the discovery of “interesting” telemetry trend relationships is bottom-up in nature, and that one of the lessons learned from the CSDL study was that telemetry charts generated in a bottom-up fashion were generally of little value because they were not designed for any purpose (see Section 7.5.4). Therefore, much research is needed in the future to control the false positive rate in data mining.

Appendix A

Software Project Telemetry Language Specification

This document describes the syntax, semantics, and design of the Software Project Telemetry Language.

A.1 Introduction

The Software Project Telemetry Language is a language that allows the user to:

- compose telemetry reports from telemetry charts,
- compose telemetry charts from telemetry streams,
- and, compose telemetry streams from software metrics using telemetry reducers and functions.

This language specification also specifies a contract for telemetry reducers and functions, but it does not prescribe what reducers and functions an implementation must provide. The relationship between the telemetry language and its reducers and functions 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 telemetry reducers or telemetry functions using the telemetry language. In other words, telemetry reducers and functions must be supplied by the language implementation.

A.2 Getting Started

This section uses one detailed example to illustrate the essential features of the software project telemetry language. It strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with a quick introduction to the language.

The following example is used throughout this section:

```
streams FilteredModuleCoverageStreams(filterMode, threshold) = {
    "Filtered Module Level Coverage Information",
    Filter(WorkspaceCoverage("Percentage", "**", "line"),
           "SimpleDelta", filterMode, threshold)
};

y-axis PercentageYAxis() = {
    "Percent", "double", 0, 100
};

chart ModuleCoverageChart(filterMode, threshold) = {
    "Module Unit Test Coverage",
    (FilteredModuleCoverageStreams(filterMode, threshold),
     PercentageYAxis())
};

report ModuleCoverageReport(threshold) = {
    "Modules with Most and Least Favorable Coverage Change",
    ModuleCoverageChart("Top", threshold),
    ModuleCoverageChart("Bottom", threshold)
};
```

This example defines a telemetry report consisting of two charts: one for the modules in a project with most significant increase in test coverage, and the other for the modules with most significant decrease in test coverage during a specified period of time.

Note that this example utilizes *WorkspaceCoverage* telemetry reducer and *Filter* telemetry function. They are used here for illustration purposes only. The telemetry language only specifies a contract that each reducer and function implementation should observe, and a syntax for how they are invoked. The availability and behavior of individual reducer and function are entirely dependent on the particular implementation your are using.

A.2.1 Telemetry Report

A telemetry report is a named set of telemetry charts that can be generated for a specified project over a specified time interval.

```
report ModuleCoverageReport(threshold) = {
    "Modules with Most and Least Favorable Coverage Change",
    ModuleCoverageChart("Top", threshold),
    ModuleCoverageChart("Bottom", threshold)
};
```

The example defines a telemetry report called *ModuleCoverageReport*, which is composed of two telemetry charts. The title of the report is *Modules with Most and Least Favorable Coverage Change*. The definition utilizes one parameter called *threshold*, which allows the user to substitute the number modules the constituent charts should display at the report invocation time. Note how the same variable appears in the chart reference section: *ModuleCoverageChart("Top", threshold)* and *ModuleCoverageChart("Bottom", threshold)*.

A.2.2 Telemetry Chart and Y-axis

A telemetry chart is a named set of telemetry streams that can be generated for a specified project over a specified time interval.

```
chart ModuleCoverageChart(filterMode, threshold) = {
    "Module Unit Test Coverage",
    (FilteredModuleCoverageStreams(filterMode, threshold),
     PercentageYAxis())
};
```

The example defines a telemetry chart called *ModuleCoverageChart*. The title of the chart is *Module Unit Test Coverage*. The definition utilizes two parameters: *filterMode* and *threshold*.

You can think of a telemetry chart as a multi-axis chart (a special kind of combined chart), with each sub-chart having its own vertical axis, but they all share the same horizontal axis. Simply put, a telemetry chart is composed of one or more sub-charts, and a sub-chart is defined by the combination of a *streams* reference and a *y-axis* reference. Multiple sub-charts can be defined in a comma separated list. The general syntax is:

```
(streams1, yAxis1), (streams2, yAxis2), ... , (streamsN, yAxisN)
```

The telemetry chart above consists of exactly one sub-chart as defined by (*FilteredModuleCoverageStreams(filterMode, threshold)*, *PercentageYAxis()*). The vertical axis for the sub-chart is defined below:

```
y-axis PercentageYAxis() = {
    "Percent", "double", 0, 100
};
```

The label for the axis is *Percent*. The definition does not utilize any parameter. An optional hint *double* specifies that the sub-chart contains double values. Other valid hints are *integer* and *auto*. The last two values are optional lower and upper bounds for the vertical axis.

Note, however, that a telemetry chart definition does not include the information about its horizontal axis, because such information can be inferred automatically from the time interval over which the telemetry chart is evaluated.

A.2.3 Telemetry Stream

Telemetry streams are sequences of a single type of software process or product data for a single project over a specified time interval.

```
streams FilteredModuleCoverageStreams(filterMode, threshold) = {
    "Filtered Module Level Coverage Information",
    Filter(WorkspaceCoverage("Percentage", "**", "line"),
           "SimpleDelta", filterMode, threshold)
};
```

The example defines a telemetry *streams* object called *FilteredModuleCoverageStreams*. A *streams* object is a collection of telemetry streams (i.e., zero or more). *Filtered Module Level Coverage Information* is the description, and the rest is the actual definition. Note that the definition contains a telemetry reducer invocation *WorkspaceCoverage(...)* and a telemetry function invocation *Filter(...)*.

A.2.4 Telemetry Reducer

A telemetry reducer aggregates low level software product and process data, and returns a collection of telemetry streams (a.k.a. a *streams* object). For example, suppose a metrics database contains coverage information for each source file in a project, then the telemetry reducer *WorkspaceCoverage* aggregates those metrics and returns a collection of telemetry streams, one for each module in the project.

A reducer can return any number of telemetry streams. While a *WorksapceCoverage* reducer returns multiple telemetry streams, a *Coverage* reducer returns only one single telemetry stream for the coverage information for the entire project. Reducer accepts parameters, but the number of the parameters and the meaning of them are entirely determined by the implementation of each reducer.

The evaluation result of a reducer call is a telemetry *streams* object. Telemetry *streams* objects can participate in arithmetic operations. You can add, subtract, multiply, and divide two telemetry *streams* objects, and the result is a new telemetry *streams* object. Suppose the telemetry reducer *WorkspaceCoverage* used in the example does not compute coverage directly. Instead, it only computes the number of source code lines covered and uncovered by test cases. Then

```
WorkspaceCoverage("Percentage", "**", "line")
```

can be equally written as:

```
WorkspaceCoverage("NumberOfCoveredLines")
/ ( WorkspaceCoverage("NumberOfCoveredLines")
    + WorkspaceCoverage("NumberOfUncoveredLines") )
* 100
```

A.2.5 Telemetry Function

A telemetry function takes a telemetry *streams* object as input, and returns another (usually different) telemetry *streams* object as output.

```
Filter(WorkspaceCoverage("Percentage", "**", "line"),
      "SimpleDelta", filterMode, threshold)
```

The example illustrates the use of a telemetry function called *Filter*. Recall that *WorkspaceCoverage(...)* is a telemetry reducer invocation, and the evaluation result is a telemetry *streams* object. This *streams* object is fed to the *Filter* telemetry function as input, so that only the telemetry streams we are interested in are returned.

This is an example where a telemetry function is used to reduce the number of telemetry streams in a *streams* object. There could also be telemetry functions that add new telemetry streams. For example, suppose you want to apply 6-sigma methodology to establish statistical control bounds for your metrics, then you can imagine a *StatisticalControlBound* telemetry function that adds two more streams: one for the upper control bound, and the other for the lower control bound.

A.3 Grammar

This chapter defines the lexical and syntactic structure of the Software Project Telemetry Language. The grammar is 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 telemetry language. The lexical processing consists of reducing the telemetry language input into a sequence of tokens which become the input to the syntactic analyzer. 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 y-axis chart report draw

operator: one of
        = + - * /

punctuator: one of
        , ; ( ) { } "

identifier:
```

```

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

string-constant:
    anything enclosed in double quotes

number-constant:
    [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 ;
    | y-axis-statement ;
    | chart-statement ;
    | report-statement ;
    | draw-command ;

streams-statement:
    streams identifier ( variables )
    = { streams-description , streams-definition }

streams-description:
    string-constant

```

```

streams-definition:
    expression

y-axis-statement:
    y-axis identifier ( variables )
    = { y-axis-definition }

y-axis-definition:
    y-axis-label
    | y-axis-label , number-type
    | y-axis-label , number-type , lower-bound , upper-bound

y-axis-label:
    identifier
    | string-constant

number-type:
    'integer' | 'double' | 'auto'

lower-bound:
    number-constant

upper-bound:
    number-constant

chart-statement:
    chart identifier ( variables )
    = { chart-title , sub-charts }

chart-title:
    string-constant

sub-charts:
    sub-chart-definition
    | sub-charts , sub-chart-definition

sub-chart-definitions:
    ( streams-reference , y-axis-reference )

streams-reference:
    identifier ( variables-and-constants )

```

```

y-axis-reference:
    identifier ( variables-and-constants )

report-statement:
    report identifier ( variables )
    = { report-title , report-definition }

report-title:
    string-constant

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

chart-reference:
    identifier ( variable-and-constants )

draw-command:
    draw identifier ( constants )

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:
    number-constant
    | ( expression )
    | function-or-reducer-call

function-or-reducer-call:
    identifier ( parameters )

parameters:

```

```

    parameter
  | parameters parameter

parameter:
    expression
  | identifier
  | constant

variables:
    <NULL>
  | variable
  | variables , variable

variable
    identifier

constants:
    <NULL>
  | constant
  | constants , constant

constant:
    number-constant
  | string-constant

variables-and-constants:
    <NULL>
  | variable-and-constant
  | variables-and-constants , variable-and-constant

variable-and-constant:
    variable
  | constant

```

A.4 Special Considerations

A.4.1 Arithmetic Operations

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

- Between two telemetry *streams* objects

The arithmetic operation is valid so long as the two *streams* objects have the same number of telemetry streams, the data points in those telemetry streams are all derived from the same time interval, and there is a way to match individual telemetry stream in the two *streams* objects. Arithmetic operations are carried out between the individual data point in the corresponding telemetry streams.

- Between one telemetry *streams* object and one number constant

The arithmetic operation is always valid. Each data point in the telemetry streams participates in the operation with the constant individually, and the result is a new telemetry *streams* object.

A.4.2 Telemetry Reducers and Functions

All reducer invocations take two implicit arguments: project and time interval. They are not covered in this telemetry language specification. It's up to the implementation to determine how such information should be passed to reducers.

Syntactically, there is no difference between a telemetry reducer invocation and a telemetry function invocation. Semantically, there is a difference. The input to a telemetry reducer is a string array, and the output is a telemetry *streams* object. The input to a telemetry function is a telemetry *streams* object, and the output is another telemetry *streams* object. It is this difference that provides the basis distinguishing a reducer call from a function call, since (1) there is no way for the user to specify a telemetry *streams* object directly using the language, and (2) reducers do not take *streams* object as argument. In the following example:

```
reducer_or_function_call_1(
    reducer_or_function_call_2(
        reducer_or_function_call_3(arguments)
    )
)
```

it goes without question that *reducer-or-function-call-1* and *reducer-or-function-call-2* are telemetry function invocations, and *reducer-or-function-call-3* is a telemetry reducer invocation.

Appendix B

Classroom Survey

This appendix presents the survey I distributed to the software engineering students in the classroom study. The classroom setting, the context, and the result of this survey are 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.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						
2.	After Hackystat sensors are installed and configured (i.e. ignoring installation and configuration effort), there is no overhead collecting metrics.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						
3.	It's simple to invoke pre-defined Hackystat telemetry chart/report analyses.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						
4.	Hackystat telemetry analyses have shown me valuable insight into my and / or my team's software development process.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						
5.	Telemetry analyses have helped me improve my software development process.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						
6.	If I was a professional software <u>developer</u> , I will want to use telemetry analyses in my development projects.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						
7.	If I was a project <u>manager</u> , I will want to use telemetry analyses in my development projects.	1	2	3	4	5	N/A
	<i>Comment:</i> _____						

 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.*

Appendix C

CSDL Data Summary

This appendix provides a summary of the data I gathered in the CSDL study. The raw data themselves cannot be published because of privacy issues. The summary is intended to provide a mechanism for the reader to “audit” my conclusions in some sense. The CSDL setting, the context, and the result of the study are reported in Chapter 7.

2006-01-05-1: I discussed the existing analysis of issue metrics with the project manager in an interview. He did not utilize the metrics in project management, because the analysis was inadequate for release cycle planning and tracking.

2006-01-05-2: I interviewed two developers on their opinions about the utility of the metrics currently collected in the lab. One of them thought the issue related metrics were not that useful, because they were quite different from what he had anticipated.

2006-01-06-1: I discussed the current status of issue management with a developer. He told me that most of his development activities were not recorded in the issue database. He suggested that we could put an issue number in commit log to link the issue tracking system and code repository together. He wanted telemetry analysis to provide him information about how much time he spent on each issue.

2006-01-09-1: I held a discussion with a developer, who estimated that only 20% - 30% of his development activities were tracked by the issue database. None of the Hackystat v6 to v7 transition issues were recorded, they were assigned in weekly meetings. None of Zorro related issues were recorded. He never followed the issue priority in resolving issues. He had five open issues on

average. When asked about his perception of the utility of the metrics currently collected in the lab, he told me that in general the metrics were useful, because it was a big leap from knowing nothing to know something. However, FindBugs reported too many problems, and that a lot of them were false-positive.

2006-01-11-1: I asked a developer about his perceptions about the utility of the metrics currently collected in the lab. He told me that metrics from FindBugs were not useful, but they could be made useful if the false-positive problem could be resolved. He also told me that he did not really understand how issue metrics were computed. When asked about issue management status in CSDL, he estimated that less than 15% of his development activities were tracked by the issue database. He told me that most of his issues were assigned through emails instead of the issue management system.

2006-01-12-1: Integration build failed. A developer forgot to test the changes before committing the code, causing JUnit failure in Core_Installer module.

2006-01-16-1: A number of topics were covered in the weekly status meeting. The project manager discussed possible changes to improve the issue management practice with the developers. The developers discussed the metrics from FindBugs and PMD, and they all appeared to agree that there were too many false-positive warnings. The developers noted that one could generate a lot of telemetry charts using the telemetry language, but they raised the question: “*do we really care about all those charts?*”

2006-01-19-1: Integration build failed. A developer modified the build script but forgot to update the server build environment, causing JUnit failures in multiple modules.

2006-01-19-2: I interview the project manager. He told me that as part of corrective measures, he went through all Jira issues and tagged them with realistic fix version numbers to get ready for the new release cycle (i.e., Release 7.3). He was identified as responsible for a recent integration build failure. I asked him how it might impact his local quality assurance practice. He told me that it was “*very effective*” in making him think more about the integration build result when committing changes.

2006-01-19-3: As part of corrective measures, the project manager sent out an email requiring that all future commits should have issue Id in commit log comment field.

2006-01-20-1: The project manager sent out an email imagining new insights could be gained after the changes made to CSDL issue management practices.

2006-01-21-1: A developer installed a Jira plug-in to enable one way link from Jira issues to SVN commits.

2006-01-23-1: The project manager sent out an email stating that he was committing in branches. This was the first time a branch was used in CSDL and was officially acknowledged.

2006-01-23-2: CSDL had a weekly status meeting. This was the first time in my observation that the new issues assigned in the meeting were recorded in the issue management system.

2006-01-24-1: Integration build failed. A developer updated the build script to invoke a newly developed sensor but forgot to install the sensor on the build server.

2006-01-26-1: Integration build failed. A developer changed code in Sdt_FileMetric module, causing JUnit failure in dependent App_Pri module.

2006-01-29-1: I enhanced the Jira sensor to collect missing information required for issue tracking telemetry analyses.

2006-01-31-1: Integration build failed. A developer updated the build script to invoke a newly developed sensor but forgot to set the proper environment variable on the build server.

2006-01-31-2: I interviewed a developer on the impact of the integration build failure alert mechanism. He commented that it made him “*a little bit more cautious*” when committing changes.

2006-01-31-3: I interviewed a developer on the impact of the integration build failure email alert mechanism. He commented: “*you might be identified as a culprit, (which) tends to make you try a little harder to think about when you actually do the thing (i.e., committing changes).*”

2006-01-31-4: I noticed an inconsistency in telemetry charts: there was no data from one of the developers. It turned out it was caused by a bad server-side project configuration.

2006-02-01-1: The same developer told me he had fixed the problem, but the inconsistency still existed in telemetry charts. The project was still mis-configured despite the developer’s effort to fix it.

2006-02-02-2: I showed some module level coverage charts to a developer. He noted that the automatically-scaled y-axis made comparison across related charts difficult.

2006-02-02-3: I interviewed a developer on the impact of the integration build failure alert mechanism. He told me that his behavior was changed significantly from "*I just build the module to see if it works*" to "*I build the entire system and test every time before commit.*"

2006-02-02-4: I interviewed a developer on the impact of the integration build failure alert mechanism. He told me that it had not changed his behavior because he was always careful about local quality assurance.

2006-02-03-1: I made the issue tracking charts available on the telemetry wall, and showed them to a developer. He commented that they could be used not only to track issue status but also to predict system release date.

2006-02-05-1: I gathered a list of the types of metrics collected in CSDL, and generated telemetry charts to show how these metrics changed over time in different grain size. One scene was devoted to project level and individual level release cycle issue tracking. The telemetry wall was up.

2006-02-06-1: CSDL had a weekly status meeting. I introduced the telemetry wall. The project level issue tracking charts were found "*highly useful*" by the project manager, but the individual level issue tracking charts were completely useless. One developer commented: "*I would rather look into Jira directly, because I don't know which issues remain.*" The project manager commented: "*I would not be interested in those individual charts.*" I noticed that the developers were using some of the charts to assess whether the underlying sensors data seemed correct or not. Further discussion identified two common causes for incorrect sensor data: (1) sensor not working correctly, and (2) bad server-side project configuration. However, in general, most of the charts were not useful. The project manager and the developers gave me a list of questions that they wished telemetry charts could help shed light on, such as some notion of quality indicators for each module in the project. There also appeared to be a performance problem with the telemetry wall. Some of the charts covering long-term metrics trends could take several minutes to show up, which was more than the developers were willing to wait.

2006-02-07-1: Integration build failed. A developer forgot to check code format before committing the changes, causing Checkstyle failure in Sdt_WorkspaceMap module.

2006-02-09-1: Integration build failed. A developer changed code in Core_Kernel module, causing JUnit failure in dependent Sdt_Activity module.

2006-02-09-2: CSDL deployed FindBugs and PMD sensors, even though the false-positive issue had not been addressed.

2006-02-10-1: Integration build failed. A third party tool (FindBug) invoked during the integration build failure terminated abnormally.

2006-02-13-1: Integration build failed. A developer changed code in Core_Kernel module, causing compilation failure in dependent App_Ggqm module.

2006-02-13-2: I wanted to display some telemetry charts on the telemetry wall in the middle of the weekly status meeting, but encountered severe server timeout.

2006-02-14-1: Integration build failed. A developer changed code in Core_Kernel module, causing JUnit failure in dependent App_Course module.

2006-02-14-2: I discussed with a developer about the current status of dependency metrics collected in the lab. He commented that the metrics was not useful. He also mentioned that a visiting scholar once tried to visualize the dependency information by using a graph, but nobody found the graph useful.

2006-02-18-1: Integration build failed. A developer forgot to test the changes before committing the code, causing JUnit failure in Sensor_XmlData module.

2006-02-20-1: While redesigning telemetry charts for the telemetry wall, I noticed several charts for module-level telemetry trends were overly cluttered.

2006-02-21-1: Integration build failed. A developer changed code in Core_Kernel module, causing compilation failure in dependent App_Mds module.

2006-02-21-2: I discussed the top-down designed charts with three of the developers, and they thought the charts displayed useful information. However, they commented that some of the charts displaying module information were too cluttered to be useful. One of them also noted that most of the lines were “uninteresting” because they represented inactive modules.

2006-02-22-1: I discussed the top-down designed telemetry wall, especially the module level quality indicator telemetry scenes, with the project manager. He liked them. However, one minor in-

convenience was that we had to check the vertical axis while comparing coverage trends in different modules.

2006-02-23-1: One of the developers toggled the telemetry wall in auto-update mode. The server stopped responding under the heavy load, and its CPU usage stayed at 100%. I had to restart the server.

2006-02-24-1: Integration build failed. I changed code in Core_Telemetry module, causing JUnit failure in dependent App_Cgqm module.

2006-02-25-1: The project size measure in CSDL was changed from SLOC to LOC. Telemetry charts were used to show the trends of the two measures, indicating that apart from absolute number changes there was no change in the trends using either measure.

2006-02-26-1: Telemetry charts showed missing coverage data. It turned out that the sensor configuration file was not updated when a developer added a new module.

2006-03-01-1: I reviewed the “DailyProjectCoverage” code, and found it that it never released references to sensor data instances, which was temporary data structure as far as “DailyProjectCoverage” is concerned.

2006-03-03-1: I generated Structure 101 reports on Hackystat modules, and discussed the metrics with the project manager. When asked whether the metrics appeared to be consistent with his intuition about the complexity of individual module, he responded *“I don’t know what to think. If it’s a metric for easiness of maintenance, then I think the module I wrote is the easiest one to maintain.”*

2006-03-05-1: Telemetry charts showed missing issue metrics. It turned out that a developer forgot to update the project configuration when adding a new module.

2006-03-05-2: The project manager created a Jira issue, requesting me to perform an audit on all daily project code to ensure references to temporary data structures were released.

2006-03-07-1: After receiving complaints about missing coverage data, the project manager sent me an email asking whether it would be possible to design charts to help detect sensor malfunction.

2006-03-08-1: I interviewed a developer on the impact of the integration build failure alert mechanism. He told me that he had spent more time on local quality assurance than before. There was

overhead, but it was acceptable since it would be much more troublesome to have integration build failures.

2006-03-08-2: I interviewed a developer. He seemed to control the local quality assurance overhead by reducing the number of commits.

2006-03-09-1: I added additional telemetry charts on the telemetry wall, designed to verify developer-side process metrics. Immediately, I detected that one developer had missing active time data. It turned out that the developer reinstalled the IDE but forgot to reattach the sensor.

2006-03-10-1: I started the code audit, and found several classes followed the same design pattern of “DailyProjectCoverage,” failing to release temporary data structures.

2006-03-11-1: Integration build failed. A developer changed code in Sdt_Coverage module, causing compilation failure in dependent App_Pri module.

2006-03-11-2: After I sent out my findings, the project manager elaborated the distinction between “cache” and “temporary data structure” in an email to call to the attention of all the developers.

2006-03-13-1: I interviewed a developer. He confirmed that almost all his work was tracked by the issue tracking system now.

2006-03-14-1: The server performance improved after the “temporary data structures” were properly disposed, but the improvement was not significant enough to reduce the telemetry wall response time to an acceptable range. I profiled the code with another developer, and we had a number of surprising findings. (1) When sensor data were cached in memory, over 90% of processor time was spent on string comparison in workspace code. The case-sensitive comparison was 10 times more expensive than case-insensitive comparison. (2) The sensor data evolution mechanism in some classes was implemented very inefficiently. For example, the “FileMetric” class spent 80% of processor time in the “recognizeData()” method, even if the data were already stored in the newest format. (3) The cost of reading sensor data from XML files was negligible.

2006-03-14-2: I interviewed a developer. He commented that the overhead on local quality assurance was acceptable given the consequence of integration build failures. He controlled the overhead by reducing the number of commits.

2006-03-15-1: Integration build failed. I was unable to determine the cause.

2006-03-15-2: I interviewed two more developers. They all confirmed that most of their work was tracked by the issue tracking system.

2005-03-15-3: I sent out an email to the developer mailing list reporting the performance profiling result.

2006-03-15-4: After reporting the performance profiling findings, I received an email from a former developer. He had experimented with a Berkeley XML DB back-end and found no performance improvement at all.

2005-03-15-5: In an email request for comments, I brought up the idea of introducing nested function calls to the telemetry language.

2006-03-16-1: During a discussion of telemetry charts with the project manager, I mentioned the idea of making relating charts more comparable by augmenting the telemetry language to allow a user to specify the vertical axis manually. He agreed that it would improve the usability of telemetry charts.

2006-03-16-2: A Jira issue (HACK-612) was created to implement filter functions to solve the usability and scalability problem of telemetry charts for large projects.

2006-03-17-1: The project manager was so impressed with the utility of those top-down designed telemetry charts on the telemetry wall, that he decided to devote an entire page on the Hackystat website to publish the results.

2006-03-17-2: The project manager used the telemetry wall to show the status of Hackystat development to outsider developers. For those overly-cluttered charts, he had to enlarge them to occupy all the nine screens to show the details.

2006-03-17-3: A Jira issue (HACK-616) was created to enhance the telemetry language to allow manually specified vertical axis.

2006-03-20-1: Integration build failed. A developer missed one file while committing his changes, causing compilation failure in Core_Installer module.

2006-03-20-2: The project manager reviewed the issue tracking chart after release 7.3 was finished, and reflected that the chart helped him determine whether more issues could be added to that release.

2006-03-21-1: Integration build failed. A developer committed local diagnostic code which should not be committed at all, causing JUnit failure in Sensor_CppUnit module.

2006-03-21-2: An external user, who managed his own server, reported degrading telemetry analysis performance proportional to server up-time. He had not picked up the recent fixes, but the problem reported was consistent with the effect of not releasing temporary data structure.

2006-03-22-1: I gave the project manager a list of recent failed integration builds together with their causes, and asked him to determine which ones were acceptable and which ones were not from his point of view.

2006-03-23-1: I discussed the telemetry charts on integration build failures and various software development process metrics with two of the developers. They confirmed that integration build failure was a complex phenomenon, and that it would be very hard, if not impossible, to predict the probability from the process metrics. The code issue density charts, which were computed from FindBugs and PMD metrics, were also available on the telemetry wall. The two developers told me that the charts failed to provide clue about Hackystat code quality, because they did not know the rules used by FindBugs and PMD to generate warnings. When asked about whether they invoked analyses themselves, one developer said: “*The language is the last thing I want to use. It looks complex.*”

2006-03-28-1: Integration build failed. I changed code in Core_Telemetry module, causing compilation failure in dependent App_Cgqm module.

2006-03-29-1: Integration build failed. A developer changed code in Sdt_Activity module, causing compilation failure in dependent App_PrjSize module.

2006-03-30-1: Integration build failed. It was caused by the same error as the previous day. The developer responsible for the error did not fix it in time.

2006-03-30-2: The project manager sent out an email giving statistics of Jira issues in recent release cycles.

2006-04-03-1: I held a discussion with the project manager, and we formalized the change to the telemetry language in order to allow a user to specify the vertical axis manually.

2006-04-05-1: Integration build failed. A developer changed one single line in App_Cgqm and did not test the change, causing JUnit failure in that module.

2006-04-05-2: There was a performance complaint on the daily project details analysis.

2006-04-06-1: Integration build failed. A developer changed code in Sdt_UnitTest module, causing compilation failure in dependent App_Pri module.

2006-04-06-2: In an email, one of the developers noted that there were many redundant computations in the daily project details analysis.

2006-04-07-1: Integration build failed. It was caused by the same error as the previous day. The developer responsible for the error did not fix it in time.

2006-04-07-2: One of the developers modified the “DailyProjectUnitTest” code taking advantage of its data access locality, and the result was remarkable: the analysis time for 2006-04-05 unit test metrics was reduced from 124 seconds to 6 seconds.

2006-04-07-3: I discussed the charts on the telemetry wall with the project manager. He was comparing release 7.4 issue tracking chart with the chart from the previous release cycle to make short term predictions.

2006-04-08-1: Integration build failed. It was caused by the same error as the previous two days. The developer responsible for the error did not fix it in time.

2006-04-09-1: I finished the implementation of filter functions and closed the Jira issue (HACK-612).

2006-04-09-2: I enhanced the telemetry language with “*y-axis*” construct, and closed the Jira issue (HACK-616).

2006-04-10-1: I did the same thing with the “DailyProjectFileMetric” code, and reduced the analysis time for 2006-04-05 file metrics from 180 seconds to just 1 second.

2006-04-10-2: The project manager was happy with the results. He wanted to formally document the design pattern as a best practice in the Hackystat developer guide in the summer.

2006-04-11-1: I update the telemetry wall, converting some charts to use manually-specified y-axes, and modifying the module-level coverage charts to use filter functions to show only the top 5 and bottom 5 covered modules. I showed the changes to the project manager and the developers. For the y-axis enhancement, they thought the change had improved the usability a lot, because comparisons could be made more intuitively with fixed vertical axes. For the filtered charts, they

liked the changes, but requested an additional chart to show modules with coverage that changed most.

2006-04-13-1: Integration build failed. A developer changed one single line in Sensor_Office and did not test the change, causing Checkstyle failure in that module.

2006-04-14-1: Integration build failed. A developer changed code in Sdt_Commit module, causing compilation failure in dependent App_Cgqm module.

2006-04-16-1: Integration build failed. A developer changed code in Sdt_Issue module, causing compilation failure in dependent App_Cgqm module.

2006-04-17-1: Integration build failed. It was caused by the same error as the previous day. The developer responsible for the error did not fix it in time.

2006-04-17-2: A quick poll indicated that, since three months ago the FindBugs and PMD reports were made available, none of the developers had spent over 1 hour in total reading the reports. This number was a little bit higher for the project manager, but it was only 2 - 3 hours.

2006-04-17-3: I discussed with the developers treatment options for each of the 17 types of FindBugs warnings found in hackyCore_Kernel module in the weekly status meeting. The comments from the developers indicated that they had learned a lot by going over their own code that generated the warnings.

2006-04-17-4: I revised the filter function implantation, and made the chart showing modules with coverage that changed most available on the telemetry wall. One of the developers commented that filter functions made the chart not only much cleaner but also much useful.

2006-04-19-1: Integration build failed. A developer imported Java code with wrong package names, causing the build failure.

2006-04-20-1: A developer modified Jira sensor code. Telemetry charts showed missing issue metrics. It turned out it was caused by a bug in the code.

2006-04-22-1: Integration build failed. A developer changed code in Sdt_Dependency module, causing compilation failure in dependent App_Cgqm module.

2006-04-22-2: An email from the project manager indicated he detected the same Jira sensor problem using the real-time sensor verification charts on the public Hackystat website.

2006-04-24-1: I discussed with the developers treatment options for the remaining types of FindBugs warnings found in the Hackystat source in the weekly status meeting.

2006-04-25-1: I modified the code issue telemetry chart to track the number of warnings falling into “fail” and “monitor” categories. The chart was primarily designed to be used by the project manager. I enhanced FindBugs report. Warnings in the “fail” category were highlighted in red color, and warnings in the “monitor” category were highlighted in blue color. I also modified the build script so that the developers could generate the report with single command on their workstations. This was primarily designed to be used by the developers to fix the warnings.

2006-04-26-1: During an interview, the project manager told me that his project management skill had improved a lot with respect to release cycle issue tracking and planning.

2006-04-26-2: The project manager assigned tasks for the developers to get rid of the warnings that fell into the “fail” category.

2006-05-06-1: Telemetry analysis indicated that all FindBugs warnings in the “fail” category had been eliminated, and the warnings in the “monitor” category had been reduced by more than a half. I had not received any complaint from the developers about the enhanced FindBugs report.

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] R. D. Bunker, 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.
- [5] R. D. Bunker, 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.
- [6] 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.
- [7] V. R. Basili. Software modeling and measurement: The goal question metric paradigm. Technical Report CS-TR-2956, University of Maryland, College Park, 1992.
- [8] 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.
- [9] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

- [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. Steele. *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] S. Chulani and B. W. Boehm. Modeling software defect introduction and removal: COQUALMO. Technical Report usccse99-510, USC Center for Software Engineering, 1999.
- [15] ClearCase. <http://www.rational.com>.
- [16] Clover. <http://www.cenqua.com>.
- [17] Cruise Control. <http://cruisecontrol.sourceforge.net>.
- [18] Costar. Softstart Systems, <http://www.softstarsystems.com>.
- [19] J. W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 2003.
- [20] CVS. <http://www.cvsnt.org>.
- [21] M. K. Dascalantonakis. Achieving higher SEI levels. *IEEE Software*, 1994.
- [22] T. DeMarco. *Controlling Software Projects*. Yourdon Press, New York, 1982.
- [23] M. Diaz and J. Sligo. How software process improvement helped motorola. *IEEE Software*, 1997.
- [24] R. Dion. Process improvement and the corporate balance sheet. *IEEE Software*, 10(4):28–35, July 1993.
- [25] Echo2. <http://www.nextapp.com/platform/echo2/echo>.
- [26] Eclipse. <http://www.eclipse.org>.
- [27] K. E. Emam. The internal consistency of the ISO/IEC 15504 software process capability scale. In *5th. International Symposium on Software Metrics*, 1998.

- [28] C. Fakharzadeh. CORADMO a software cost estimation model for RAD projects. In *16th International Forum on COCOMO and Software Cost Modeling*, 2001.
- [29] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Publishing, 1997.
- [30] 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.
- [31] FindBugs. <http://findbugs.sourceforge.net>.
- [32] 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.
- [33] B. G. Glaser. *Doing Grounded Theory - Issue and Discussion*. Sociology Press, 1998.
- [34] B.G. Glaser and A. L. Strauss. *A Discovery of Grounded Theory - Strategies for Qualitative Research*. Sociology Press, 1967.
- [35] R. B. Grady. Practical software metrics for project management and process improvement. *Prentice-Hall*, 1992.
- [36] R. B. Grady and D. L. Caswell. Software metrics: Establishing a company-wide program. *Prentice-Hall*, 1987.
- [37] 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.
- [38] J. Henry. Personal software process studio. <http://www-cs.etsu.edu/psp>, 1997.
- [39] W. Hetzel. *Making Software Measurement Work: Building an Effective Software Measurement Program*. QED Publishing, 1993.
- [40] W. S. Humphrey. Characterizing the software process. *IEEE Software*, 5(2):73–79, March 1988.
- [41] W. S. Humphrey. *Managing the Software Engineering*. Addison-Wesley, 1989.
- [42] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

- [43] W. S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.
- [44] W. S. Humphrey, T. R. Snyder, and R. R. Willis. Software process improvement at hughes aircraft. *IEEE Software*, 1991.
- [45] ISO. Iso 9001 quality systems - model for quality assurance in design/development, production, installation and servicing. *ISO*, 1987.
- [46] ISO. Iso/iec tr 15504-1: 1998 information technology - software process assessment. *ISO*, 1998.
- [47] JavaCC. <http://javacc.dev.java.net>.
- [48] JBlanket. <http://csdl.ics.hawaii.edu>.
- [49] Jira. <http://www.atlassian.com>.
- [50] JMeter. <http://www.apache.org>.
- [51] P. M. Johnson. You cann't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashville, TN, December 2001.
- [52] P. M. Johnson and A. M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6):85–88, November 1998.
- [53] 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.
- [54] P. M. Johnson and M. G. Paulding. Understanding HPCS development through automated process and product measurement with Hackystat. In *Second workshop on productivity and performance in high-end computing*, February 2005.
- [55] JUnit. <http://www.junit.org>.
- [56] 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.

- [57] H. Kou and P. M. Johnson. Automated recognition of low-level process: A pilot validation study of Zorro for test-driven development. In *Proceeding of the 2006 International Workshop on Software Process*, May 2006.
- [58] P. Kulik and M. Haas. Software metrics best practices - 2003. *Technical report, Accelerate Research Inc.*, 2003.
- [59] P. Kuvaja, J. Simila, L. Krzanik, A. Bicego, G. Koch, and S. Saukonen. *Software Process Assessment and Improvement: the BOOTSTRAP approach*. Blackwell Publishers, 1994.
- [60] F. Latum, R. Solingen, M. Oivo, B. Hoisl, D. Rombach, and G. Ruhe. Adopting gqm-based measurement in an industrial environment. *IEEE Software*, 1998.
- [61] LOCC. <http://csdl.ics.hawaii.edu>.
- [62] Christoph Lofi. Continuous GQM. Master's thesis, Technical University Kaiserslautern, Germany, 2005.
- [63] S. B. Merriam. *Qualitative research and case study applications in education*. Jossey-Bass Publishers, 1998.
- [64] M. B. Miles and A. M. Huberman. *Qualitative data analysis: A sourcebook of new methods*. Sage Publications, 1994.
- [65] C. A. Moore. Project LEAP: Personal process improvement for the differently disciplined. In *International Conference on Software Engineering*, pages 726–727, 5 1999.
- [66] 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.
- [67] W. C. Peterson. SEI's software process program - presentation to the board of visitors, 1997. Software Engineering Institute, Carnegie Mellon University.
- [68] S. L. Pfleeger and C. McGowan. Software metrics in the process maturity framework. *The Journal of Systems and Software*, 1990.
- [69] PMD. <http://pmd.sourceforge.net>.
- [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] W. Royce. *CMM vs. CMMI: From conventional to modern software management*. The Rational Edge, 2002.
- [74] SEI. *The Capability Maturity Model: Guidelines for Improving Software Process*. Addison-Wesley, 1995.
- [75] R. Solingen and E. Berghout. *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.
- [76] R. Solingen, E. Berghout, and F. Latum. Interrupts: Just a minute never is. *IEEE Software*, 1998.
- [77] Standish. The Chaos Report. http://www.standishgroup.com/sample_research/PDFpages/-chaos1994.pdf, 1994.
- [78] Visual Studio. <http://www.microsoft.com>.
- [79] SVN. <http://http://subversion.tigris.org>.
- [80] R. Tesch. *Qualitative research: Analysis types and software tools*. Falmer, 1990.
- [81] Google Web Toolkit. <http://code.google.com/webtoolkit>.
- [82] D. Tuma. Software process dashboard. <http://processdash.sourceforge.net>, 2000.
- [83] H. T. Wolcott. *Writing up qualitative research*. Sage Publications, 2001.
- [84] Cost Xpert. Cost Xpert Group Incorporated, <http://www.costxpert.com>.