

LEARNING EMPIRICAL SOFTWARE ENGINEERING USING THE SOFTWARE
INTENSIVE CARE UNIT

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

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

DECEMBER 2009

By

Shaoxuan Zhang

Thesis Committee:

Philip M. Johnson, Chairperson

Henri Casanova

Scott Robertson

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Information and Computer Sciences.

THESIS COMMITTEE

Chairperson

©Copyright 2009

by

Shaoxuan Zhang

To my mom, dad, wife, and my new born baby girl Ruby.

Acknowledgments

This research would not be possible without the following people who have provided me with guidance, support, and encouragement along the way.

Abstract

In software engineering, the importance of measurement is well understood, and many efficient software development metrics have been developed to help measurement. However, as the number of metrics increase, the effort require to collect data, analyze them and interpret analysis results quickly becomes overwhelming. This problem is even more critical in educational approaches regarding empirical software engineering.

The Software Intensive Care Unit is a new approach to facilitate software measurement and control with multiple software metrics. It use the Hackstat system to achieve automated data collection and analysis, then uses the collected analysis data to create an intensive monitoring interface for multiple “vital signs”. A vital sign is a wrapper of a software metric with proper presentation. It consist of a historical trend and a newest state value, both of which is colored according to its “health” state. I hypothesize that the Software ICU is helpful for students to study empirical software engineering.

My research deployed and evaluated Software ICU in a senior-level software engineering course. Students’ usage was logged in the system, and a survey was conducted. The results provide supporting evidence that Software ICU does help students in course project development and project team organization. In addition, the results of the study also discover some limitations of the system, including inappropriate vital sign presentation and measurement dysfunction.

Table of Contents

Acknowledgments	v
Abstract	vi
List of Figures	ix
1 Introduction	1
1.1 The Problem	2
1.2 Software Intensive Care Unit Approach	2
1.3 Evaluation of Software ICU	4
1.4 Thesis Statement	4
1.5 Thesis Structure	5
2 Related Work	6
2.1 TSP/PSP	6
2.2 Research Based on Automated Data Collection	7
2.2.1 Project ClockIt and Retina	8
2.2.2 PROM	9
2.3 Previous Case Studies of Hackystat	11
3 Hackystat	13
3.1 Hackystat Framework	13
3.1.1 Sensors	13
3.1.2 SensorBase	14
3.2 Analysis Services	15
3.2.1 Daily Project Data Analysis	15
3.2.2 Telemetry Analysis	15
3.3 Project Browser	16
4 Design and Implementation of The Software ICU	17
4.1 Interface of Software ICU	17
4.2 Definition of Related Terms	17
4.3 Vital Signs	18
4.4 Vital Sign Presentation	21
4.4.1 Stream Trend Coloring	22
4.4.2 Participation Coloring	23
4.5 Mini Chart Drill-Down	23
4.6 Vital Sign Configuration	24
4.7 System Customization	26
5 Classroom Evaluation	28
5.1 Case Study in Classroom	28

5.2	Experimental Limitations	29
6	Results	31
6.1	Feedback regarding Hackystat system	31
6.2	Verification of System Usage	32
6.3	Utilities of Vital Signs	34
6.4	Vital Sign Popularity	35
6.5	Feasibility in a professional software development context	37
6.6	Thesis Statement Revisited	38
7	Conclusions	39
7.1	Contributions	39
7.2	Future Direction	39
A	2008 Classroom Evaluation Questionnaire of Hackystat	41
B	Results form 2008 Classroom Evaluation Questionnaire of Hackystat	44
	Bibliography	61

List of Figures

<u>Figure</u>	<u>Page</u>
1.1 An example medical ICU screen.	3
1.2 An example Software ICU screen.	3
2.1 Progression of PSP	7
2.2 ClockIt BlueJ Data Visualizer summary	8
2.3 Data viewers of Retina.	9
2.4 Architecture of the PROM system	10
2.5 Screenshot of course project to date analysis of Hackystat in 2003	11
2.6 Screenshot of file-metric telemetry analysis of Hackystat in 2006	12
3.1 Architecture of Hackystat	14
4.1 A screenshot of Software ICU	18
4.2 The Vital Sign Configuration panel in Software ICU	25
6.1 SICU usage on per student bias	32
6.2 Analysis count on a per-student basis	33
6.3 Vital sign popularity from survey	34
6.4 The final states of all class projects in Software ICU.	34
6.5 Usage of Telemetry Analyses	36

Chapter 1

Introduction

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software[1]. As a famous software engineering researcher says, “you can neither predict nor control what you cannot measure”[2]. Measurement is an indispensable step to help software development achieve a state characterized by predictable and controllable processes. Though lots of research and literature exists on software metrics, some of their limitations including measurement distortion and dysfunction are well known, and a major error in management decision support comes from using a single metric in isolation. In order to overcome these limitations, the use of measurement from multiple dimensions is necessary to obtain a more comprehensive perspective on any given software attribute of interest, such as readability, maintainability, modifiability, reliability and so forth.[3][4]. However, manipulating multiple software metrics is not simple. The increased effort required for data collection and analysis needs to be addressed to reduce the overhead of measurement. Selection of metrics and their presentation also demands careful consideration to prevent metric data from overwhelming the user and preventing useful application.

In this research, I use concepts from the medical intensive care unit (or medical critical care unit), where multiple vital signs are monitored in an automatic and efficient manner. Medical ICUs provide a set of “vital signs” that help doctors determine when a patient’s health is stable, improving, or declining. Using the medical ICU as a metaphor, I built an application called the Software Intensive Care Unit. It provides an intensive monitor interface with multiple vital signs, which are software metrics that wrapped with proper presentation to indicate their “health” states. However, the design of the Software ICU is a great challenge. Both what data to present and how to present them are essential design decisions, but neither is well-studied. The variety of development

settings makes this problem even more complicated. But I am not confident that there exists a golden rule for all situations. Hence, providing a capability for configuration and customization is important. In this research, I tune the Software ICU to the scenario of course project development in undergraduate classroom.

1.1 The Problem

In software engineering, the importance of measurement is well understood, and many efficient software development metrics have been developed to help measurement. However, as the number of metrics increase, the effort required to collect data, analyze them and interpret analysis results quickly becomes overwhelming. This problem becomes even more critical when introducing software measurement to a software engineering course, where students are still struggling to make the transition from programming to software development. There are so many things (such as system design, code style, software quality control, collaboration, etc.) they need to focus on that utilizing software measurement is usually found to be a distraction. This leads to the impressions that software measurement is difficult.

1.2 Software Intensive Care Unit Approach

The Software Intensive Care Unit (Software ICU) is based upon the Hackystat system, which already provides automated data collection and analysis, and further helps developers to interpret software measurement results and control the software development process. The Software ICU adopts the metaphor of a medical intensive care unit, where a set of vital signs are intensively monitored to determine the health state of the patient, then treatment is planned according to the state. [Figure 1.1](#) illustrates an example medical ICU screen. Each vital sign in a medical ICU represents the condition of an organ system. A vital sign within its normal range of behavior indicates that the corresponding organ system functions normally. When a vital sign departs its normal range of behavior, it is an alarm of possible organ system failure, and treatment may be required to keep the organ functional. When this happens to more vital signs, the patient's health state is more critical, and emergency treatment is required to avoid death.

In the Software ICU, software metrics are used as vital signs, and are monitored intensively. The "health" states of the software are determined, and marked with color, so that developers can plan "treatment" for their software project accordingly. [Figure 1.2](#) illustrates an example Soft-



Figure 1.1. An example medical ICU screen.

Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
hackystat-analyse-dailyprojectdata (2)	85.0	2.5	8.9	2.0	2518.0	0.7	1.0	7.0	286.0
hackystat-analyse-telemetry (2)	81.0	3.5	9.1	2.0	6523.0	0.7	1.0	6.0	500.0
hackystat-sensor-ent (4)	27.0	2.9	5.3	18313.0	2004.0	5.5	46.0	57.0	78.0
hackystat-sensor-eclipse (2)	N/A	3.1	N/A	N/A	817.0	0.1	N/A	4.0	8.0
hackystat-sensor-manual (1)	39.0	N/A	N/A	N/A	N/A	0.0	N/A	2.0	4.0
hackystat-sensor-shell (2)	54.0	2.5	6.8	2.0	1153.0	1.6	1.0	12.0	40.0
hackystat-sensorbase-uh (2)	83.0	3.0	9.4	162.0	4496.0	2.9	4.0	15.0	279.0
hackystat-ui-wicket (4)	52.0	2.0	11.2	1043.0	8018.0	4.2	30.0	62.0	296.0
hackystat-utilities (4)	72.0	1.9	4.0	2.0	1286.0	0.3	1.0	16.0	200.0

Figure 1.2. An example Software ICU screen.

ware ICU screen. Each metric represents a factor in software development process. When a metric departs its reasonable behavior, it indicates that a factor of the software development process might go wrong. “Treatment” is required to fix that in order to avoid possible project failure.

Similar to medical ICU, Software ICU’s vital signs are presented with both historical trend and current state, each of which is then colored separated. Different vital signs may use different coloring methods and parameter configurations. Unlike the medical ICU, whose vital signs (temperature, respiration, etc.) have been studied in some cases for hundreds of years, no comparable body of research exists in software engineering on how to intensively monitor multiple software measurements to determine the state of software project or how different behavior of metrics impacts upon the state of software project. Therefore, my selection of vital signs and their configurations are research hypothesis, and are validated in the case study.

1.3 Evaluation of Software ICU

Undergraduate students enrolled in a Software Engineering course assisted with the evaluation of this research. The class consists of 18 students. In the second half of the semester, they were divided into 5 groups and developed two course projects. Hackystat and Software ICU were introduced to the class to help them understand the health state of a software project.

They used the system for approximately six weeks, and their activities on the system were logged during that period. At the end of the semester, the students were invited to participate in a survey that asked their opinion of Hackystat and Software ICU.

I compared and analyzed the result from system logs and questionnaire responses to find out how they used the system and what impact the system had on their development.

1.4 Thesis Statement

This research investigates the mechanism and technology of Software ICU and gathers data to assess the following hypotheses:

1. Adopting metaphor of medical ICU to software engineering is practical and feasible.
2. The selection of vital signs is appropriate.
3. The coloring mechanism correctly illustrates the health state of the vital signs.

4. Knowledge of health state of their projects helps students improve their performance in collaborative software development.

The first hypothesis claims that it is possible to implement an application that monitors multiple software development measurements and can be used to direct software development practice in a way similar to medical ICU.

The second hypothesis claims that the selection of vital signs is adequate to reveal potential defects during software development.

The third hypothesis claims that with a decent coloring method, the vital sign of different conditions will be assigned different colors, and the same color can be traced back to similar conditions.

The fourth hypothesis concerns a chain reaction of events. When students know the health state of the vital signs of their projects, they will need to fix their code or improve their development practice if the vital sign is not healthy. By trying to keep vital signs healthy, students should discover better ways to collaborate with other teammates and produce high-quality software.

1.5 Thesis Structure

The remainder of this thesis is as follows. Chapter 2 presents some previous studies related to this research. Chapter 3 describes the Hackstat system, which Software ICU is built upon. Chapter 4 contains detailed description of the design and implementation of Software ICU. The evaluation procedures are described in Chapter 5 and the results are discussed in Chapter 6. Finally, Chapter 7 contains the conclusions and future directions of this research.

Chapter 2

Related Work

This chapter presents some work related to my research.

The first part discusses previous research on empirical software engineering concepts. Most previous research on measurement-based software engineering focuses on methodology. Effective approaches are developed and deployed in actual practice. However, the lack of automation adds significant overhead to developers, thus leading to the impression that they are hard to do. Research on Hackstat and the Software ICU is oriented towards a new generation of approaches to PSP metrics that automate data collection and analyze[5].

The second part discusses three recent research projects that focus on automated data collection. Two of them mainly focus on introductory level programming courses and are not very suitable to senior software development or professional settings. The third one is very similar to Hackstat and has related industry studies.

The last part discusses two previous related case studies of the Hackstat system to provide some insight into the use of Hackstat in a classroom setting prior to the use of the Software ICU.

2.1 TSP/PSP

The Personal Software Process (PSP)[6] and the Team Software Process (TSP)[7] are among the most extensively studied approaches for measurement-based software engineering. They were developed by Watts Humphrey to teach students (in university and industry alike) about the use of large scale methods based on the Capability Maturity Model (CMM)[8]. The PSP attempts to scale down industrial software practices to fit the needs of small scale program development. Software processes and software engineering disciplines are gradually introduced through small

program projects (e.g. course assignment projects). The PSP maturity progression is shown in **Figure 2.1**. Students gather both process and product measures on a set of projects. By comparing the measurement result to their original planning, they gain insight into their programming habits, both pros and cons, and improve their process to higher level of maturity.

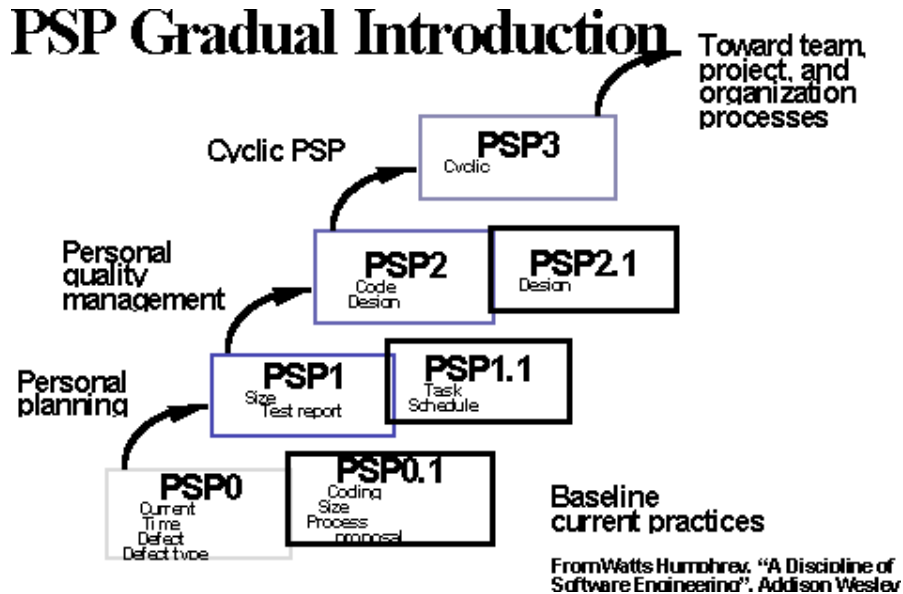


Figure 2.1. Progression of PSP

A major drawback of the PSP is lack of automation. Developers have to manually record their process and product data (mostly the development time and number of defects). The high overhead of data collection raises a barrier to its introduction and adoption. Additionally, it is not easy to “digest” the data. Developers have to manually analyze their logged data in order to understand their performance, then be able to improve it.

On the contrary, Software ICU explores how one can provide a higher level of automation in tracking and analyzing software process and product data.

2.2 Research Based on Automated Data Collection

Project ClockIt and Retina are two recent research projects based on automated data collection to support entry-level programming courses, while PROM is the most similar research to Hackystat.

2.2.1 Project ClockIt and Retina

Project ClockIt provides a data logger as a BlueJ¹ extension. It records developer's open/close of project and package events, file change and delete events and compilation results. Data is saved to a local file and later sent to a database via the Internet. A data visualizer integrated into BlueJ is available to view data about the current project. Figure 2.2 shows an example of this visualizer. Data stored in database is used for statistic analysis such as class averages. A web interface is also available to instructors to view the individual data of their students and class average analysis data.

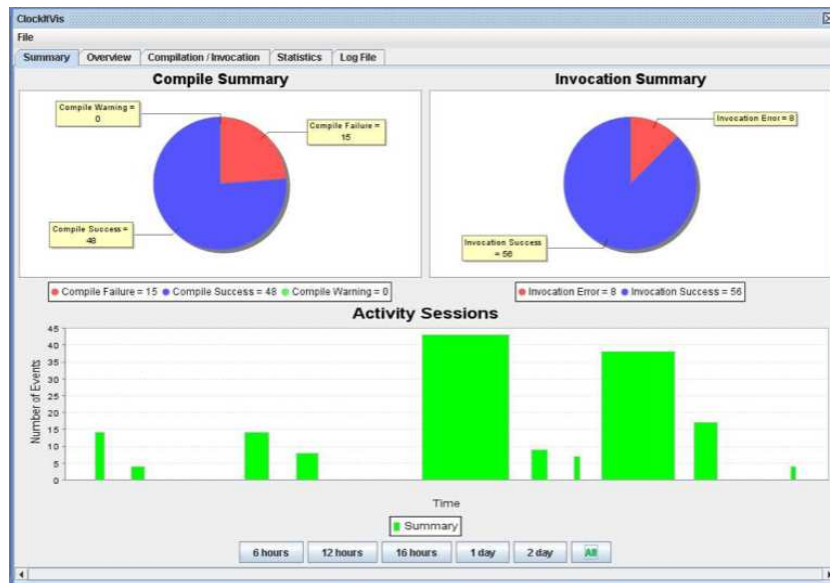


Figure 2.2. ClockIt BlueJ Data Visualizer summary

Closely related to ClockIt, Retina also provides automated data collection. Though Retina provides more tool support (BlueJ, Eclipse and command-line compiler), it focuses on a even smaller area of programming events: compilation. It gathers data from students' compilation events, mostly compilation errors. In addition to its data viewer (see Figure 2.3), it also provides a recommendation tool for students. The tool uses instant messaging (IM) to give students an estimate of the amount of time required for the upcoming assignment, and the compilation errors one is likely to make. These are based on both the student's previous data and the data from courses of previous semesters.

¹"BlueJ is an integrated Java environment specifically designed for introductory teaching." –Quoted from <http://www.bluej.org/about/what.html>

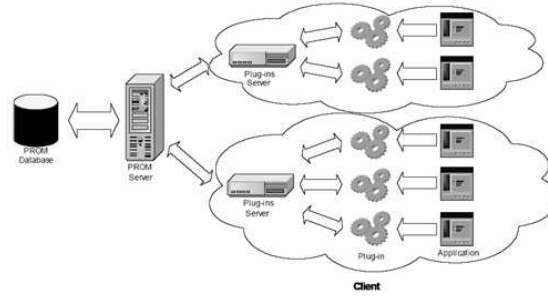


Figure 2.4. Architecture of the PROM system

individual, detailed data, the leader has access to the aggregated data of the whole team, and the manager has access to project level aggregated data.

Compared to Hackstat, PROM's data is stored as analyzed metrics results while Hackstat stores the raw sensor data. The disadvantage of storing raw data is that, analysis has to be executed each time the results are requested, while the advantage is that the abilities to modify analysis algorithms and to run new analysis on existed data are retained. Moreover, Hackstat's caching mechanism compensates the disadvantage to some extent. In PROM, different data viewers are provided to different groups of users while Hackstat does not restrict views based upon the role of a member on a project.

A recent case study of PROM in an industrial environment[10] discusses the lessons learned from two years experience of using the PROM system in the IT department of a large company in Italy. Evidence indicates that adopting a system like PROM requires a long set-up phase and needs the company and development team's patience and commitment to succeed, but it can eventually delivers value to the company.

One of the lessons suggests that data presentation is as important as data accuracy and simplicity, brevity and clarity is preferable. Another lesson suggests that fast aggregated view of data is desired, and users of different roles favor different aggregations, e.g. developers like reports of their daily activities, while team leader and manager like summary views of data on team and project level. The Software ICU's simple and fast data presentation and high configurability and extensibility would appear to address these requirements.

2.3 Previous Case Studies of Hackystat

The classroom study presented in this thesis is the third case study of the Hackystat system in a classroom setting.

The first case study was performed in 2003 used an early version of Hackystat[11]. During that time, Hackystat was only collecting 4 types of metrics (Active Time, Size, Unit Tests and Coverage). The system was oriented around a set of “Course” analyses that were tailored to an educational setting. Those analyses summarized the individual team project metric data in tabular form, and also presented comparisons of all of the course projects (Figure 2.5). The case study evaluation showed that the installation of Ant sensors is the most significant barrier to the system. It was too difficult to install without direct help from the development team. But the overhead of use is relatively low and analyses were usable and useful. However, the lack of data privacy was uncomfortable for some students.

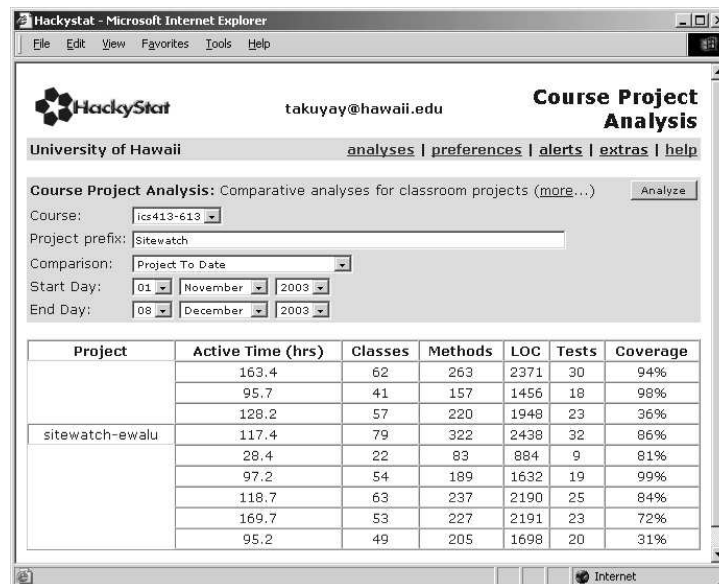


Figure 2.5. Screenshot of course project to date analysis of Hackystat in 2003

The second case study was performed in 2006 as a partial replication of the first case study[12]. Hackystat had undergone significant change from 2003 to 2006. The sensor installation, which is the major barrier to the system in 2003, was automated by the hackyInstaller GUI, which greatly lowers the overhead of configuration for developers. The evaluation confirms this with a substantial drop in sensor installation difficulty. However, a new sophisticated Telemetry analy-

sis **Figure 2.6** and its complex user interface raised the difficulty of using it and interpreting data, leading to slight drop in usability and professional feasibility.

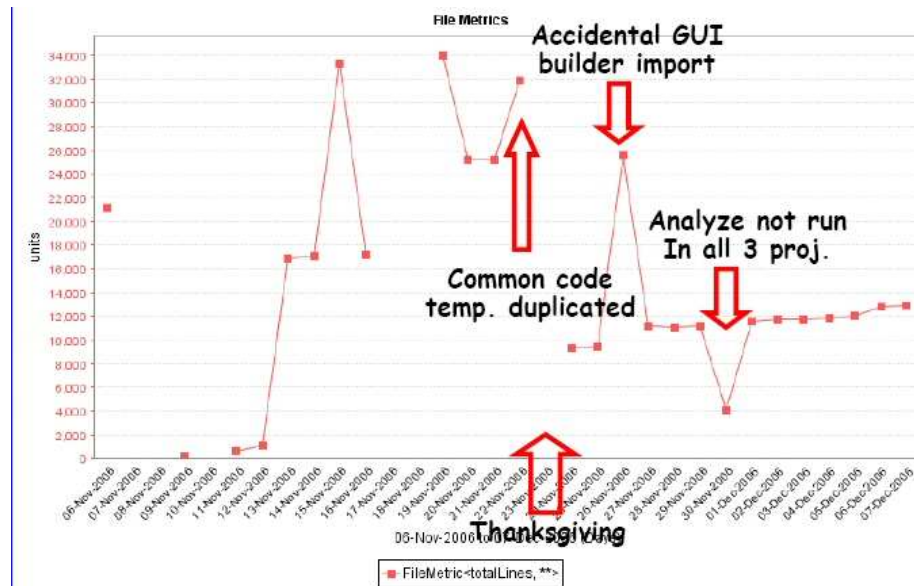


Figure 2.6. Screenshot of file-metric telemetry analysis of Hackystat in 2006

In 2007, Hackystat was re-implemented with a new architecture. Adopting a service-oriented architecture enables the development of multiple user interfaces separate from the data collection and analysis components. The Software ICU is built upon a new web-based UI called Project Browser, and the classroom study is also based on this user interface.

Chapter 3

Hackystat

In this research, the Software ICU is built upon Hackystat to fulfill automated data collection and analysis. This chapter briefly describes the Hackystat system, which was invented by Professor Philip M. Johnson, in the Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawaii at Manoa.

3.1 Hackystat Framework

Hackystat is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data. Hackystat consists of many software services that communicate using REST architectural principles[13]. These software services can be categorized into 4 groups: sensors, data repository, analysis services and viewers. [Figure 3.1](#) shows the architecture of Hackystat system.

3.1.1 Sensors

Sensors are small software plugins that collect data from the use of tools and applications. Currently, sensors are available for many development software systems including Eclipse, Emacs, Ant, etc. Sensor data is represented in XML, and consist of seven basic elements: data owner, resource, timestamp, runtime, tool, Sensor Data Type and properties. The first six are required and the last one is optional.

The Sensor Data Type(SDT) is specified for every piece of sensor data when collected, so that the same type of data can be collected from different tools and higher level services can easily determine which data is relevant to them. Sensor data is designed to record only an piece of atomic data such as size of a single file. The runtime field is used to group data that belongs to the same

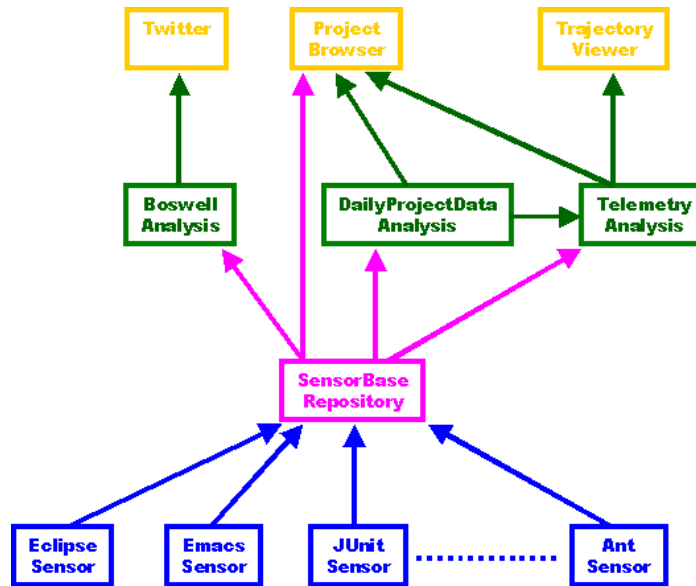


Figure 3.1. Architecture of Hackystat

event, such as the file metrics associated with a project. Properties are additional information for different types of sensor data, such as coverage value for coverage SDT and lines of code for file metric.

Sensors are designed to work automatically without any attention from the user apart from initial configuration. In order to reduce internet communication and support offline work, data is temporarily stored locally, then sent to the data repository every several minutes or when internet connection is available.

3.1.2 SensorBase

Sensor data is sent to the data repository, called the SensorBase. The SensorBase stores the data as it is sent from sensors, and provides RESTful interface for easy manipulation of the data using HTTP. Sensor data can be queried with the six required elements mentioned above via HTTP calls, and data is sent back as XML. The SensorBase is implemented with a database manager abstract class, thus it is easy to add support for different database implementations. The current version of Hackystat provides database support for Derby, Oracle and PostgreSQL.

3.2 Analysis Services

The analysis services of Hackystat provide abstractions of the raw data from SensorBase. DailyProjectData and Telemetry are the two most fundamental analysis services of Hackystat.

3.2.1 Daily Project Data Analysis

As its name indicates, DailyProjectData (DPD) service provides abstractions of sensor data associated with a single project within a 24 hour period, which represents a simple software development metric on a single day. Data for a single project includes data from all members of that project. In a DailyProjectData instance, both a summary value, e.g. total development time across the project, and detailed values, e.g. total development time of a project member, are available. So it is easy for higher level services to use this data.

Each DPD analysis generates software metric from data of a certain Sensor Data Type. Current available DPD analyses are Build, Code Issue, Commit, Complexity, Coupling, Coverage, Dev Time, File Metric, Issue, and Unit Test. These DPD analyses are the basis of the Hackystat system, most other analysis services are based on DPD. While DPD is a low level of abstraction, these can also be considered as the available software metrics in Hackystat.

3.2.2 Telemetry Analysis

Based on DPD service, the Telemetry service provides abstraction over a longer period of time such as several days, weeks or months. A Telemetry Chart consists of one or more streams of data points. Each data point represents the metric value of in a single granularity (day, week or month). Together they show the trend of the metric(s).

There is a special group of Telemetry charts called Member-Level Telemetry. These charts consist of several stream, each of which belongs to a project member. They are used in Software ICU's drill down feature to compare performance of each member within a project.

To support the work practices of different organizations, Telemetry service provides a domain specific language that allows to build new Telemetry Chart with Telemetry stream lines. The predefined Telemetry Charts are all written with this language.

Telemetry streams can also accept parameters to refine the object data. This feature is inherited in Software ICU, where user can configure the parameters of each Telemetry analysis of each vital sign (more detail discuss in [section 4.3](#) and [section 4.6](#)).

3.3 Project Browser

Project Browser is one of the viewers in Hackystat system. It is based on Wicket¹, a Java-based web application framework. Project Browser is integrated with viewers to all Hackystat services, which are organized as tabs.

With the help of Wicket's modularization, viewers on Project Browser can share common panels, such as project/date selection panels and Ajax loading process panel, which facilitate the development of new pages. This also makes user's experience more consistent across different viewers. Therefore it now serves as a data presentation and high level analysis development center. Several new presentations and high level analysis are developed upon it. The Software ICU is one of them.

¹<http://wicket.apache.org/>

Chapter 4

Design and Implementation of The Software ICU

In order to utilize multiple software development metrics to manage software development process, I adopt the metaphor of the medical ICU and develop a system called the Software Intensive Care Unit (Software ICU). It consists of a set of vital signs, each of which is based on one software development metric and indicates the project's "health" state from one perspective.

4.1 Interface of Software ICU

The interface to the Software ICU is separated into two parts. The left-hand side is the control panel, where the user can pick the analysis period, data granularity and selected projects to analyze. The right-hand side consists of three panels: the data panel, the loading process panel, and the configuration panel. Each of these panels is discussed in following sections. The data panel is the major panel that shows the result of SICU analysis. [Figure 4.1](#) shows an example of Software ICU.

4.2 Definition of Related Terms

Before the discussion of the design of Software ICU, two terms are needed to be clarify.

"health"

"vital sign"



Figure 4.1. A screenshot of Software ICU

4.3 Vital Signs

Similar to the medical ICU, the use of multiple software development metrics in Software ICU is necessary because there is not a single metric that can determine the health state of a software project. Similar to medical vital signs, each software metric shows different aspect of a software project. Changes in one of them may or may not indicate a change in the overall health state, but changes in more of them indicates a higher possibility that health state changed. In this study, we use nine vital signs in the Software ICU.

Vital signs of software projects are measured by various software development process or product metrics. Each of these vital signs reveals an aspect of the health state of the software project. In this section I will discuss all these vital signs.

Coverage Coverage is a good indicator of the tests' quality. It stands for the test coverage of source code in unit testing, which is usually measured as the percentage of code units (line, method, class, etc) that is executed during testing. There are a number of coverage criteria, such as line, method, class, conditional, etc. In the Software ICU, the user can select which to use. No matter which criteria is chosen, higher coverage is always better because a higher

percentage of code covered by unit testing indicates a lower risk of defects in the untested code. However, high coverage does not necessarily mean good quality unit test, or vice versa. One reason is that, in some situations, it is difficult to achieve high test coverage because of the difficulty of verifying results, especially when using UI frameworks. Another reason is that the code executed during unit testing can be unverified. For example, when testing an image processor with a given image file, the code of loading the image file is executed, but the test does probably not have assertion about the correctness of loading the file. But as long as developers don't have the intent to trick coverage in order to pretend to be writing enough unit tests (which is possible if coverage is misused to judge their performance), raising coverage is always a good thing.

Cyclomatic complexity Cyclomatic complexity, a measurement of the complexity of a program developed by Thomas J. McCabe, measures the number of linearly independent paths through a program's source code[14]. The higher the cyclomatic complexity, the more distinct control paths exist in a program module, and the more difficult it is to achieve high path test coverage. Additionally, code of high complexity is often difficult to understand, thus it is hard to maintain. Therefore, program modules with low complexity are preferred. But high complexity is not necessarily evil. The nature of some programs just requires a high level of complexity. Also a raise in complexity is sometimes unavoidable during development, especially when optimizing code performance. However, developers should try to avoid high complexity unless it is necessary, especially in early stages of development, so that the code is easier to maintain in future.

Coupling Coupling, or dependency, is the degree to which each program module relies on one or more of the other modules. It is a measurement of the complexity of the whole system's module reference tree. Whenever one module is modified, there will be a chance that the changes may cause bugs in one of modules that relies on it. Therefore, higher coupling implies higher risk of introducing bugs when making changes, thus harder to maintain. High coupling might also be harder to reuse because dependent modules must be included. Therefore, Coupling is suggested to be kept low.

DevTime DevTime, an abbreviation of Development Time, is a measurement of the time spent on development tasks by developers. Hackystat uses a special approach to measure this: for each 5 minute interval, if any development activities are observed by the tool sensors, the developer is considered to be developing during that interval. It relaxes the criteria of measuring

development time so that coding while reading from documentation will get the same DevTime as intensive coding period. However, Hackstat sensors for DevTime are only available in several IDEs (currently available to Emacs, Eclipse, and Visual Studio). No sensors are available for other applications that might be used during developing, such as browsers, E-mail clients, office systems, or other editors/readers. So the monitored development activities are limited. Moreover, some development activities, such as reading and learning, are very difficult to track. Therefore, DevTime should not be simply used to determine a developer's effort. But if the habit of an individual does not change a lot, then the DevTime of a developer should be relatively stable over time. Thus large sudden increase in DevTime is a possible sign of bad developing habit like "start late near deadline".

Churn Churn is a measurement of the changes (addition, deletion and/or modification) of code that is made into repository. It is usually measured by LOC (lines of code). It is an indicator of developers' contribution to the project. Interpretation of this metric is depends upon the stage of development. In the early stages of development, churn is expected to be high because new code is being added. During the maintenance of a system, churn is mainly from fixing bugs and adding new features, both of which are fewer for a stable system, thus churn is expected to be lower. In terms of development behavior, the churn of developers reflects to some extent the amount of work they are doing. It tends to be relatively stable over time in the same project because the work rate of an individual does not vary a lot in the same coding condition. Dramatic change in churn of an individual developer while DevTime not changing respectively is a bad phenomena, which might due to bad developing habit like "copy and paste without understanding".

Commit Commit measures the number of commitments made into repository. "Commit early, commit often" is a well-accepted guideline of continuous integration. For the same amount of churn, more commits implies better following of this discipline.

Size The size of the project is measured by the source lines of code (SLOC), which counts the number of lines in the text of the program's source code. It can be a sign of the effort put into the project, However, SLOC alone does not make as much sense about the state of the project as Churn. We include this vital sign only to give user an idea of the size of the project, just like the height in your medical record.

Test Test is a count of unit test tasks invoked in a period of time. Unit testing is a software verification and validation method in which a developer tests individual units of source code. It is used to ensure that code meets its design and behaves as intended. A requirement of good development behavior is to test while coding, or even better, use “Test Driven Development” (TDD). No matter what development pattern you follow, unit testing is an indispensable step and regular execution of unit tests is always a good sign of a healthy development habit.

Build Build is a count of the times a build system (such as Ant, Make, or Maven) is invoked in a period of time. A build task accomplishes necessary steps to ensure the correctness of the code before commit. It typically consists of compilation, code inspection, unit testing, documentation generation, etc. It is a usual activity in software development nowadays. Though how often to build largely depends on personal preference and habit, it is advised to build often to ensure the correctness of the system.

These nine vital signs are the default set in the Software ICU, but this can be changed. Users can determine which vital signs to use, as well as creating new vital sign analysis with Telemetry charts. More detail about this configuration and customization is provided in [section 4.6](#) and [section 4.7](#).

4.4 Vital Sign Presentation

As reported in a case study of PROM, data presentation is as important as data accuracy[10]. One of our primary goals for the Software ICU is to provide a proper presentation to help interpret large amount of software metrics data. In order to achieve this goal, the Software ICU uses mini charts to integrate historical data and uses color to categorize the health state of a vital sign.

A vital sign analysis consists of two parts: a numerical latest value and a mini historical chart.

Latest Value represents the newest state of the vital sign in the analysis period. In our implementation, it shows the most recent associated DailyProjectData. If there is no DPD on the latest date of the analysis period, it will search back the time period for the first available data of that DPD. The latest value will be “N/A” only when there is no any data of that metric in the whole analysis period.

Mini chart represents the trend line of the associated metric data over the analysis period. This mini chart is implemented as bar charts. Each bar represents the DailyProjectData value of

the metric on a unit of granularity (day, week or month). Bars heights are scaled so that the highest bar is almost reach the top of the chart.

However, providing the last values and mini charts does not completely address the requirement for fast data interpretation. Thus we further enhance the representation by adding colors to those numerical values and charts to provide intuitive idea of the “healthy” state of the vital signs.

Generally, we use color green to represent a “healthy” state, red to represent a “unhealthy” state, and yellow for an uncertain state. This color pattern is good for indicating states because it matches conventions people attach to color and thus most people can understand it without reading instructions.

Different vital signs may use different coloring methods, and the latest value and the mini bar are colored separately. The choice of coloring method mainly depends on the nature of the vital sign. In general, vital signs that have clear preference of higher or lower, like most based on software development product metrics (Coverage, Complexity, Coupling) will use Stream Trend Coloring method, and vital signs based on software process metrics will likely to use Participation Coloring method. Sometimes, there may be no ideal coloring method for a vital sign, such as FileMetric, then the user can select to leave that vital sign uncolored.

4.4.1 Stream Trend Coloring

The Stream Trend Coloring method determines the health of a metric by its value and trend. It colors the latest value as well as the mini chart. It takes three parameters: *HigherBetter*, *Higher Threshold* and *Lower Threshold*. Users can decide the preferable trend, higher or lower, using the *HigherBetter* parameter. For example, a rising mini chart is considered to be good if the *HigherBetter* parameter is set to true. A trend is considered to be rising if there is no value point lower than the one before, and if the last value is greater than the first value. A falling trend is determined in the opposite way. In order to be able to categorize trends that have some small disruption as raising or falling, the Stream Trend Coloring method considers small amounts (proportional to the average of the first and the last value) of change as equal. Stable trends are always considered as “healthy” because in that case it is as good as “healthy” that user doesn’t need to pay much attention to it, while the actual value will be shown in the latest value where the value will be judged to be “healthy” or not. And unstable trend is marked as yellow because it is no fast way to tell if it indicates a good state or not.

Higher Threshold and *Lower Threshold* parameters are only used when coloring the latest value. Values exceeds the higher threshold will be colored green if *HigherBetter* is true, or red if *HigherBetter* is false. Values lower than the lower threshold are colored in similar way. Values between these two thresholds are always colored yellow.

4.4.2 Participation Coloring

The Participation Coloring method determines the health of a stream by the participations of all members in the project. It only colors the mini chart, leaving the latest value uncolored. This coloring method is designed to detect the health state of team cooperation, mainly via software process metrics. It takes three parameters: *Member Percentage*, *Threshold* and *Frequency*. The Participation Coloring method colors a mini chart green if

1. there are more percentage of members than the *Member Percentage* parameter that,
2. have the metric value greater than or equal to the *Threshold* parameter per day,
3. for more frequently than the *Frequency* parameter in the analysis period.

A mini chart is colored yellow if it does not meet the green requirement, but the metric of the team as a whole meets the requirement of green, i.e.,

1. the combined metric value is greater than or equal to the *Threshold* parameter per day,
2. for more frequently than the *Frequency* parameter in the analysis period.

If the yellow requirement is not meet neither, the mini chart will be colored red.

In other words, Participation Coloring method colors a vital sign green if most members of a project are making noticeable contribution to the project regularly, and color it yellow if the vital sign does not achieve the green state but there is someone making contribution to the project in most of the time, and color it red otherwise, which means, in terms of this vital sign metric, the contribution of members to the project is rare and/or insignificant.

4.5 Mini Chart Drill-Down

In each non-empty mini chart, the Software ICU provides a link to the drill-down Telemetry analysis. The drill-down Telemetry analysis is the analysis that used to generate the mini chart. For most of software product metrics, such as Coverage, Complexity and Coupling, the drill-down

Telemetry analysis will show the same chart as the mini chart in Software ICU’s vital sign block, just in different style with more detailed axes. However, for software process metrics, instead of the original chart, an associated member-level Telemetry analysis is shown in the drill-down.

The member-level Telemetry shows multiple stream lines in the chart, each of which represents the analysis of a member of the project. From this member-level Telemetry, it is easy to see members’ participation to the project in this metric. This is most useful when combined with Participation Coloring in the Software ICU, where you see the summary result of members’ participation, and then understand the detail with member-level Telemetry analysis.

The drill-down Telemetry analysis uses the same parameters as used in Software ICU, thus the non-member-level chart should be identical with the one in Software ICU. Vital signs with drill-down to member-level Telemetry, use as well the member-level Telemetry to generate the mini chart in vital sign block by summing all stream lines into one. Software ICU provides different integrating method to handle vital signs that use member-level Telemetry, more detail is discussed in [section 4.7](#).

4.6 Vital Sign Configuration

The Software ICU provides the user with access to the configuration of each vital sign. User can enable/disable a vital sign, choose its coloring method, and configure the parameters of the associated Telemetry analysis.

By clicking the “Show Configuration” button in the input panel on the left, user can open the configuration panel to its right. Then the “Show Configuration” will be disabled when the configuration panel is shown. [Figure 4.2](#) shows an example of the vital sign configurations. The first column is the name of the vital sign with the checkbox to enable or disable a vital sign. When a vital sign is disabled, the configuration of color method and Telemetry parameters will disappear, however, the settings are not discarded, thus when enabled again, it will be the same as before it was disabled. The second column is the color method. The current version of Software ICU provides three choices: StreamTrend, Participation and None. By choosing the first two, its associated parameters, which are discussed in [section 4.4](#), are shown next to the drop-down selection field. When “None” is selected, nothing will be shown in that space. The last column is the Telemetry parameters, which is defined in the definition of Telemetry charts, and will be directly transferred to Telemetry service when retrieving Telemetry analysis for vital sign presentation. Because of the implementation, the results of enabling/disabling a vital sign and selecting different color method

Software Project Portfolio Analysis

From Date: 2009-10-02 17

To Date: 2009-10-06 17

Granularity: Day

Show Configuration

Project(s):

- ☐ AmbientHackstat
- ☒ Default
- ☐ Hackstat
- ☐ hackstat-analysis-d
- ☐ hackstat-analysis-te
- ☐ hackstat-sensor-ant
- ☐ hackstat-sensor-ecl
- ☐ hackstat-sensor-em
- ☐ hackstat-sensor-exa
- ☐ hackstat-sensor-ma
- ☐ hackstat-sensor-sho
- ☐ hackstat-sensor-vim
- ☐ hackvstat-sensor-xm

OK Cancel

Measure Name	Color Method	Higher Threshold	Lower Threshold	Higher Better	Telemetry Parameters
<input checked="" type="checkbox"/> Coverage	StreamTrend	90	40	<input checked="" type="checkbox"/>	mode: Percentage granularity: method
<input checked="" type="checkbox"/> Complexity	StreamTrend	0	0	<input checked="" type="checkbox"/>	mode: AverageCc threshold: 10 tool: JavaNCSS
<input checked="" type="checkbox"/> Coupling	StreamTrend	20	10	<input type="checkbox"/>	coupling: All mode: Average type: class threshold: 10 tool: DependencyFinde
<input checked="" type="checkbox"/> Churn	StreamTrend	900	400	<input type="checkbox"/>	cumulative: <input type="checkbox"/>
<input checked="" type="checkbox"/> Size(LOC)	None				sizemetric: TotalLines tool: *
<input checked="" type="checkbox"/> DevTime	Participation	Member(%) 50 Threshold 0.5 Frequency(%) 50			cumulative: <input type="checkbox"/>
<input checked="" type="checkbox"/> Commit	Participation	Member(%) 50 Threshold 1 Frequency(%) 50			cumulative: <input type="checkbox"/>
<input checked="" type="checkbox"/> Build	Participation	Member(%) 50 Threshold 3 Frequency(%) 50			result: * type: * cumulative: <input type="checkbox"/>
<input checked="" type="checkbox"/> Test	Participation	Member(%) 50 Threshold 10 Frequency(%) 50			mode: TotalCoun cumulative: <input type="checkbox"/>
<input type="checkbox"/> CodeIssue					

OK Reset to Default Configuration Instructions

Figure 4.2. The Vital Sign Configuration panel in Software ICU

will be saved immediately, but other fields will only be saved when the “OK” button in the bottom is pushed. When the “OK” button is pushed, the configuration panel will disappear after setting is saved, and the “Show Configuration” button will become available again.

In order to persist the user’s configuration setting between each visit, the configuration settings are saved in server side using UriCache. UriCache is a wrapper around the Apache JCS system¹. It is designed to provide an API well suited to the needs of Hackystat services. The vital sign configuration objects are directly cached, under the name of the user. The cache expiration timer is set to 300 days so that it will not easily be expired. But if the cache is expired, the system will use the default setting of the vital signs.

Next to the “OK” button is the “Rest to Default” button. It will restore all vital sign configuration settings to default, and the result of restoring will be shown and saved immediately.

In the bottom-right of the configuration panel is a link called “Configuration Instructions”. When clicked, it will show a simple instruction of the configuration panel in a pop-up window.

4.7 System Customization

Beside the ability to configure vital signs on the fly, Software ICU provide also offline customization of default vital signs. All vital signs, including the default set discussed above, are defined in PortfolioDefinition XML files. There are two place the system will look for these XML files. The first place is inside the package of detail panel of Software ICU, where the default set of vital signs are defined. The other place is `/.hackystat/projectbrowser/`, where “ ” stands for user’s home directory. Here is an example of the definition XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<PortfolioDefinitions>
  <Measures>
    <Measure name="Coverage"
      classifierMethod="StreamTrend"
      enabled="true"
      telemetryParameters="Percentage,method">
      <StreamTrendParameters higherBetter="true"
        lowerThreshold="40"
        higherThreshold="90"/>
    </Measure>
    <Measure name="MemberDevTime"
```

¹“JCS is a distributed caching system written in java. It is intended to speed up applications by providing a means to manage cached data of various dynamic natures.” –<http://jakarta.apache.org/jcs/>

```

        alias="DevTime"
        merge="sum"
        classifierMethod="Participation"
        enabled="true">
      <ParticipationParameters memberPercentage="50"
                               thresholdValue="0.5"
                               frequencyPercentage="50" />
    </Measure>
    <Measure name="FileMetric"
            alias="Size(LOC)"
            enabled="true">
    </Measure>
  </Measures>
</PortfolioDefinitions>

```

There is a root element called *PortfolioDefinitions*, enclosing a single element *Measures*. Within the *Measures* element, there are a set of *Measure* element, each of which stands for a vital sign. Each *Measure* element can take up to six attributes:

1. The *name* attribute is required. It is the name of the Telemetry analysis used in this vital sign.
2. The *alias* attribute is optional. When it is set, it will be used as the name of this vital sign. Otherwise, the *name* attribute will be used as this vital sign's name.
3. The *classifierMethod* attribute defines the default coloring method, either *StreamTrend* or *Participation*. This attribute is optional. When it is unset, the default coloring method will be none.
4. The *enabled* attribute defines if the vital sign is enabled by default. If set to false, the vital sign will be disabled by default. But the user is still able to enable it in configuration panel.
5. The *merge* attribute defines the method to integrate multi-stream telemetry. It is necessary for member-level telemetries to work. "sum", "min" and "max" are available choices. If it is unset, the first stream of the telemetry will be used. Because of the order of streams in a multi-stream telemetry is not guaranteed, using member-level telemetry without setting this attribute might cause unexpected result.
6. The *telemetryParameters* attribute is the Telemetry parameters of the telemetry analysis defined in *name* attribute. It can be unset, then the default parameters will be used. This attribute accept value formatted the same way as the Telemetry Rest API, i.e. common separated values ordered the same as parameter definition in the Telemetry analysis.

The *Measure* element also can have up to two optional sub-elements. They are *StreamTrendParameters* and *ParticipationParameters*, each of which defines default parameters of the corresponded coloring method, and can exist together regardless what is set in the *classifierMethod* attribute. They take the same attributes as their parameters discussed in [section 4.4](#).

Chapter 5

Classroom Evaluation

This chapter discusses the evaluation of the Software ICU.

5.1 Case Study in Classroom

The evaluation of the research hypotheses in this project occurred in an academic environment by undergraduates in a senior-level Software Engineering course (ICS 413) at the University of Hawaii. The class consisted of nineteen students. They were gradually introduced to software engineering concepts like specification, modeling, analysis, and design, along with useful tools including the Eclipse IDE, the JUnit testing framework, the Subversion configuration management system, the Ant build system, and the Hackystat system. As part of the first 7 weeks, they were guided to the three prime directives of open source software (1. The system accomplishes a useful task. 2. An external user can successfully install and use the system. 3. An external developer can successfully understand and enhance the system.), and practiced on these directives on individual basis. Then they were divided into groups of two to work on open source projects hosted on Google Project Hosting, using Subversion system for another 3 weeks. Then Hackystat and the Software ICU were added to their practice. They continued to work on their projects in large size of groups for approximately 5 weeks.

At the end of the Fall 2008 semester, the students in ICS 413 were asked to respond to a questionnaire soliciting their opinions regarding Hackystat and the Software ICU. The complete questionnaire can be found in [Appendix A](#).

In order to eliminate the potential bias that due to the attempt, either consciously or unconsciously, to "please" the instructor/designer who would presumably be gratified by positive responses to the questionnaire, responses were provided anonymously to the course instructors. It is

done in this way: Before the questionnaire is given out, I provided each of the students a “secret” code. The correspondence between the secret codes and the students are only known by me, but not the instructor of the class. Response was optional, but the students were offered extra credit points for providing their opinions. The list of names who should be awarded extra credit was sent to the class instructor without identifying individual responses. Eighteen out of the nineteen students contacted provided responses.

In addition to the survey, I also logged students’ activities on Software ICU and the related Telemetry page of Project Browser. Every time when students run an analysis on Telemetry or Software ICU, the name of the analysis, its parameters, and the timestamp of the request is recorded in the log file. I also record the event of clicking the mini chart to run Telemetry drill-down analysis, so that I can assess the drill-down feature is actually useful. The last event I track is changing the configuration of vital signs. However, no meaningful action of configuration is record. This is reasonable because of their lack of sophistication regarding software measurement at the time of the study.

At the end of the evaluation, the log data was compared to the feedbacks from the survey to help verify students responses.

5.2 Experimental Limitations

It is important to recognize the limitations of this study. Compared to the limitations associated with previous study in 2003 and 2006, anonymity is achieved, but others are still unsolved in class evaluation.

First, this data is drawn from a limited sample size of 18 students. The subjects therefore have a relatively narrow and homogeneous background in software development.

Second, the context in which they used the system was a course project. Course projects tend to be smaller, narrower in scope, and with less pressure on the developers than an industrial context. It is one thing to get a poor grade for doing a poor job, it is another thing to lose your job for doing a poor job. In addition, students are not working full-time on the system; the development project is just one assignment among several.

These are all major limitations on the external validity of the responses. They do not make the results meaningless, but rather help provide a perspective on how to gain additional evidence in future that would confirm/disconfirm these initial findings. For example, it would be helpful to

deploy Hackystat in a real software company, and then gather data anonymously from the coders and managers. Other insights into future research directions will be covered in an upcoming section.

Chapter 6

Results

The results of the classroom evaluation questionnaire can be found in [Appendix B](#)

6.1 Feedback regarding Hackystat system

Besides the purpose of research regarding Software ICU, this study can also be interpreted as a evaluation of Hackystat's new service oriented architecture.

The responses of the questionnaire indicate that sensors installation is more difficult than Hackystat in 2006. This is not surprising because of the fact that a client-side installer package was provided in 2006, which is not yet available in the time of this study. However, once the sensors are installed correctly, no further effort is required in data collection. Because all the students are using the public services of Hackystat¹, there is no effort required in the server-side configuration, which is reported to be the biggest installation/configuration difficulty in Hackystat case study in 2006.

The sensors' installation difficulties is mainly cause by the documentation. Though installation guides are provided for every component, the documentation is too distributed to follow as a result of Hackystat's service-oriented architecture, which reduced the coupling among components, but also reduced the correspondence among components' documentation.

Regarding development data sharing, most students felt OK sharing development data with other members. But three students had concerns that sharing development data would reveal their programming habits and introduce too much competition of statical stats, which made them nervous. It is interesting that those three students are the three with least Software ICU running count in [Figure 6.2](#). It is reasonable to infer that they worked less harder than other students and did

¹SensorBase:<http://dasha.ics.hawaii.edu:9876/sensorbase>,
jectData:<http://dasha.ics.hawaii.edu:9877/dailyprojectdata>,
try:<http://dasha.ics.hawaii.edu:9878/telemetry>, ProjectBrowser:<http://dasha.ics.hawaii.edu:9879/project>

DailyPro-
Teleme-

not want to be noticed. However, this is an example of how measurement dysfunction might cause negative effect on developers.

6.2 Verification of System Usage

Figure 6.5 and Figure 6.2 show the data from system usage logging. I combine it with the data from questionnaire to confirm that the students' responses from questionnaire are reflecting the truths of their practice.

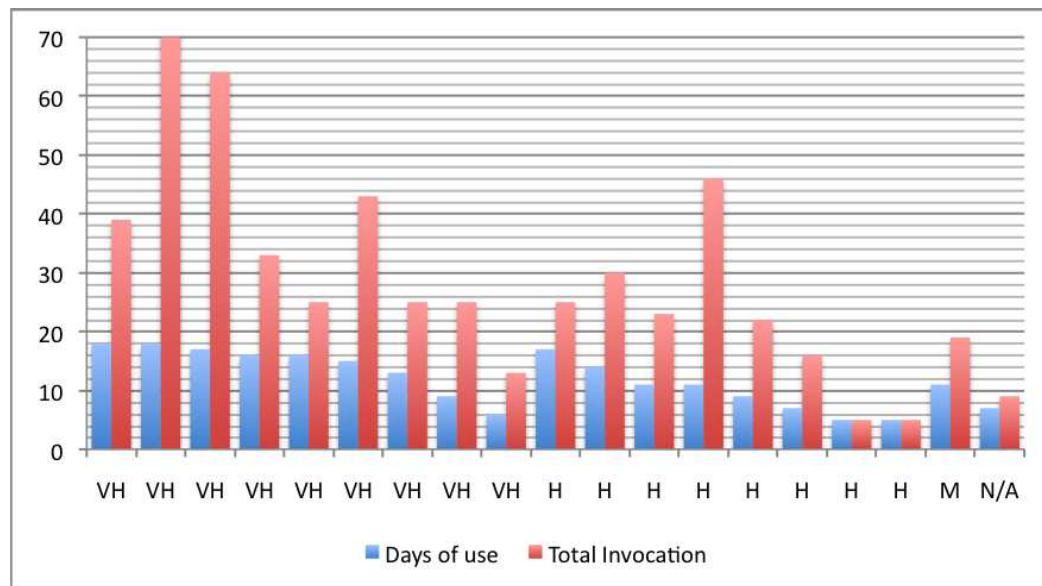


Figure 6.1. The count of days when SICU was used along with the total invocation on per student basis. Each pair of columns represents data of one student. The X axis shows the responses from questionnaire. VH = every day or more; H = 2-3 times per week; M = once a week; N/A = not available.

When verifying the questionnaire responses against the log data, we find that the choices of question “How frequently did you use the telemetry page?” and “How frequently did you use the Software ICU?” are somehow ambiguous. Though “every day or more” is surely asking how many days you use the analysis, “2-3 times a week” may be understood as times of invocations. Figure 6.1 shows data of these two interpretations. If we consider the answers as “days of use”, the actual use frequencies are much lower than reported, because there are 28 days in the evaluation period but the highest number of days of use is only 18. But if we consider the answers as “times of invocations”, the invocation frequencies are more matched to reported frequencies. However, in either case, the difference of actual usage between students who claim to use SICU analysis “every day or more”

and “2-3 times a week” is not obvious. Though the total invocation times and days of the first group is higher than the second, some students of the second group used SICU analysis more frequently than the students of the first group. But this error is acceptable because the frequency of use is just as remembered and might not be precise. So if the criteria is weakened and both “every day or more” and “2-3 times a week” are considered as “did use SICU frequently”, all responses match their log data except three of them. Those three students claimed that they use Software ICU 2 to 3 times a week or more, but actually only half as much as they claimed (The lowest one with response “every day or more” and the lowest two with response of “2-3 times per week”).

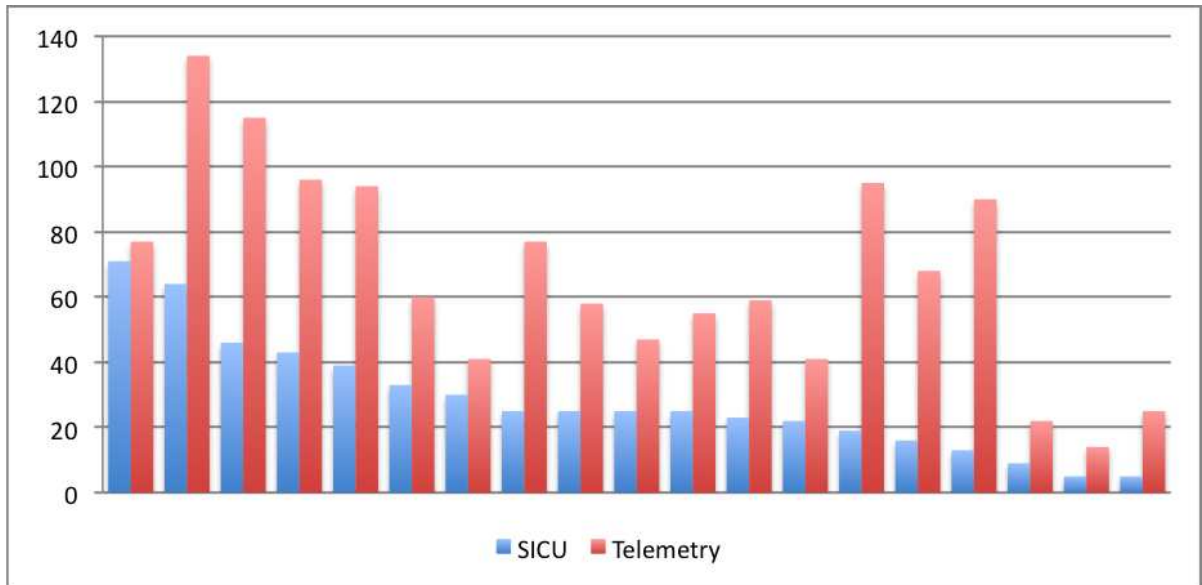


Figure 6.2. Analysis count on a per-student basis during the evaluation period. Each pair of columns represents data of one student.

I also find that though the reported frequency of SICU and Telemetry are similar, Telemetry’s analysis invocations are much more than SICU’s(see [Figure 6.2](#)). But this matches the nature of these two analyses: SICU shows the overall summary of a project’s health and no need to run more than once a day, while Telemetry shows detail of a vital sign and would often be run multiple times in every use.

Because both questionnaire responses and log data show matched evidence that students are using Software ICU and other Hackstat services frequently, I believe that students participated in this evaluations actually have plenty of use experience with the Software ICU and Hackstat and the responses of the questionnaire are based on their real experience and opinions.

6.3 Utilities of Vital Signs

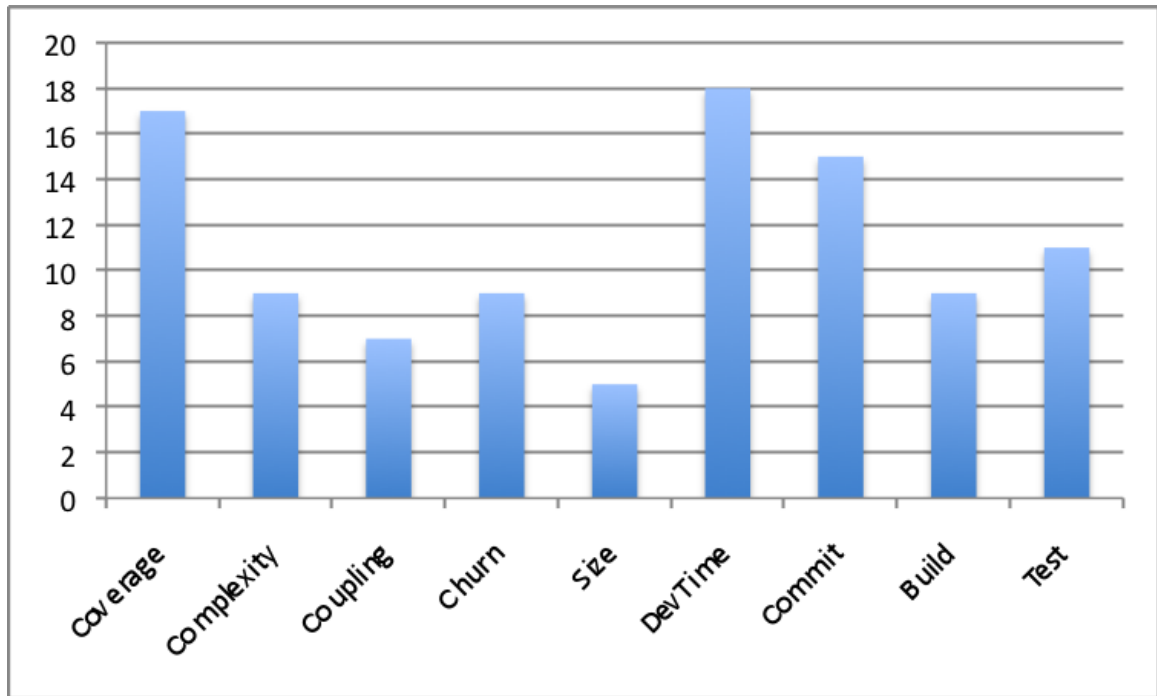


Figure 6.3. Counts of selections of each vital sign in responses of question “If you used the Software ICU, please check the vital signs that were useful to you.”.

Regarding Software ICU as a whole, 7 out of 9 vital signs are considered to be useful by at least half of the respondents (Figure 6.3). 10 out of 18 responses said Software ICU was accurately reflecting the health of their project via colors. Other 6 responses are not denying the utilities of vital signs, but are arguing that some vital signs are not accurate enough to determine a project’s health. Only one student found it is “hard to determine what will fall into green, red, or yellow”, and the last student said he failed successfully configure the sensors. Overall, students were quite positive regarding the utilities of vital signs.

Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (5)	63.0	1.6	6.9	835.0	3497.0	3.2	21.0	42.0	150.0
duedates-ahinahina (5)	61.0	1.5	7.9	1321.0	3252.0	25.2	59.0	194.0	274.0
duedates-akala (5)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0
duedates-ornaormao (5)	64.0	1.2	6.2	1566.0	5597.0	22.3	59.0	230.0	507.0
duedates-ulaula (4)	90.0	1.5	7.8	1071.0	5416.0	18.5	47.0	116.0	475.0

Figure 6.4. The final states of all class projects in Software ICU.

Three vital signs cause the most concern in student feedback: Coupling, Churn and DevTime.

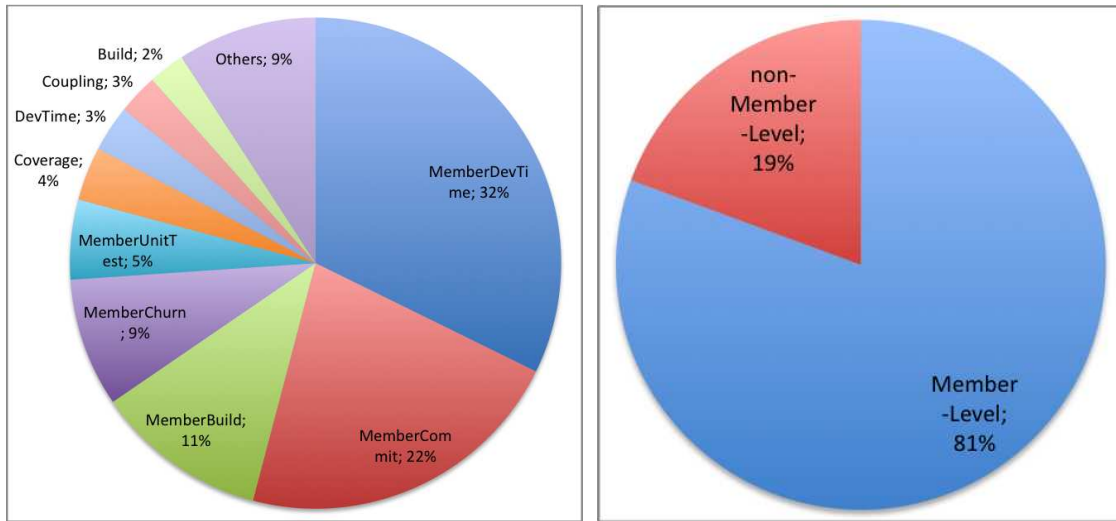
Coupling cause concern in that its increase during development is unavoidable, at least there is no easy way to avoid, especially when adopting new packages. The course is not focussed on how to design software to avoid significant increase in Coupling, and students were not experienced enough to figure it out by their own. So student felt confused about what to do: adding new classes and packages is necessary to accomplish the tasks, but it will also increase the coupling and make the vital sign to turn red. As shown in the final states of the class projects in Software ICU ([Figure 6.4](#)), 4 of 5 projects failed to keep their Coupling trend green. However, considering their other vital signs, they are not necessarily doing worse than the one with green Coupling. On the other hand, this indicates that the presentation of Coupling vital sign does not accurately reflect the true “health” state of the projects.

Churn’s concern can be ascribed to misuse of coloring method. As discussed in [section 4.3](#), Churn is preferred to be relatively stable. Neither significant increase or decrease is desirable. However, a proper coloring method is not yet implemented. Setting the default coloring method is set to StreamTrend with *HigherBetter* parameter set to false come out to be a mistake. It mislead students to the impression that Churn should be kept lower, which confused students. On second thought, without a coloring method exclusively designed for Churn, even Participation coloring method might be better than StreamTrend because it can better present process vital signs.

The problem of DevTime is its lack of completeness. Because of the limited collection of DevTime sensors, only a few applications are supported by Hackystat, and only one of them is primarily used by students: the Eclipse IDE. The effort on other development activities, such as reading books, researching online, or even pair programming is not collected. Some students felt compelled to do more coding to catch up with their group partners.

6.4 Vital Sign Popularity

The number of invocations of Telemetry analyses can be an indicator of vital sign popularity and usefulness. Both log data ([6.5\(b\)](#)) and questionnaire responses indicate that Telemetry is mainly used to run member-level analyses. The two most used analyses are MemberDevTime and MemberCommit ([6.5\(a\)](#)). In the responses to the question of vital sign usefulness ([Figure 6.3](#)), DevTime and Commit are also among the three most popular vital signs. The other one in the top-three is Coverage.



(a) Invocations of each Telemetry analysis

(b) Comparison of member-level and non-member-level analyses

Figure 6.5. Usage of Telemetry Analyses

It is not surprising that Coverage is among the most popular vital signs. Compared to other productive metrics, it is the most intuitive indicator of project's quality. It is not in the frequently used Telemetry analyses because there is no need to run a separate Telemetry analysis. Users can get all the information from the coverage vital sign in Software ICU.

But DevTime and Commit's popularities were not expected by me prior to the evaluation. Survey result indicates that, this is not a special case: vital signs based on software process metrics attract much more attention than those based on software product metrics (Figure 6.3). Popularity of process vital signs (DevTime, Commit, Build, Test, and Churn) exceed all productive vital signs except Coverage (Complexity, Coupling and Size). There are three major reasons that lead to this result.

The first reason is that popular vital signs are much easier to interpret than those that are not popular. The meaning of popular vital signs is very straight-forward. On the contrary, as mentioned in students comments, complexity and coupling metrics are more difficult to comprehend. Though the general guidelines are the lower the better, the meaning of a certain number is not easy to understand because of the nature of these metrics. Also, as the development progresses and more features and functions are added to the code, complexity and coupling always tend to increase. Ad-

ditionally, unlike coverage that one can simply “write more tests to increase the coverage”, there is no single obvious way to reduce complexity and coupling.

Size is an exception because it naturally has no preference to be higher or lower and it is the only vital sign that does not have a default coloring method. It is intended to stay in default vital signs set as a reference rather than an indicator.

The second reason is that productive metrics are less dynamic than process metrics. Productive metrics are statistics of the existed source code. The changes are usually slowly accumulated, so the change of productive metrics are relatively linear. On contrary, process metrics are measurement of human activities, which can vary a lot from day to day. A developer can code for 6 hours in one day but not code at all in the next day. Therefore, the change of process metrics is more interesting.

The last reason, as indicated in students’ responses, is because Software ICU is used by some students to improve their team’s process by tracking members’ activities. As mentioned by a student, member-level Telemetry analyses provide a quantitative way to identify who is falling behind in terms of effort output, thus team members can be more self-critical by comparing their individual data to the groups. Students do make use of these process vital signs to better organize team development. And these vital signs offer a way to motivate students to work hard.

However, one concern is that DevTime and Commit are so popular that they may also induce measurement dysfunctions that affect user’s behaviors. As noted by Austin in his *Measuring and Managing Performance in Organizations*[15], measurement dysfunctions defining characteristic is that the actions leading to it fulfill the letter but not the spirit of the stated intentions. At least one student actually experienced this negative effect. He explicitly pointed out that the quantitative measurement of their activities led to a competition of stats within the group. More students have possibly been affected as well because as indicated in that student’s answers, his team share the similar opinion of the “stats competition”.

6.5 Feasibility in a professional software development context

Responses of the questionnaire show that most students thought it was at least somewhat feasible to use Hackstat and Software ICU as a professional developer. Some comments point out some potential barriers of adopting Software ICU to professional setting, including data privacy, data completeness and measurement dysfunctions. But most comments suggest that Hackstat and the Software ICU are useful in professional setting in one way or another.

6.6 Thesis Statement Revisited

With the observations made from the evaluation results, the following summaries can be made about the three hypotheses:

1. *Adopting metaphor of medical ICU to software engineering is practical and feasible.*

The implementation of Software ICU shows abundant evidence to support this hypothesis. Critical functionalities are all implemented at the time of evaluation. Only concerns about implementation from evaluation is the requirement of a choice of emphasized layout that focus exclusively on single project, which is not difficult to implement at all.

2. *The selection of vital signs is appropriate.*

Evidence did suggest that the selected set of vital signs satisfied the need of measurement of the students, and most vital signs were consider useful. However, one concern is that, at the time of study, students' lack of sophistication regarding software measurement may make this conclusion questionable.

3. *The coloring mechanism correctly illustrates the health state of the vital signs.*

The result of this is mixed. Firstly, the concept of "color to state" is supported from students opinion. Secondly, some vital signs are thought to be correctly colored while some are not. The Participation coloring method enjoyed positive responses on all deployed vital signs, while the StreamTrend coloring method's performance is appropriate on Coverage and Complexity, but is debatable on Coupling and Churn. Lesson from Churn suggest that careful selection of coloring method is as important as development of new one.

4. *Knowledge of health state of their projects helps students improve their performance in collaborative software development.*

Comments from questionnaire imply evidence to support this hypothesis. Students stated that the health state of a vital sign guided them to discover and adjust problems in their code or team organization, which will not be (easily) noticed otherwise. But negative impression from inappropriate coloring also affects the conclusion of this hypothesis to some extent.

Chapter 7

Conclusions

7.1 Contributions

This research contributes to empirical software engineering in three ways.

The first contribution is the evidence that Software ICU metaphor and presentation help students understand and utilize software development metrics to improve their individual development performance and team collaboration.

The second contribution of this research is the insight into a new way to teach empirical software engineering course. The classroom case study reveals not only positive impact of the Software ICU paradigm, but also negative affect of measurement dysfunction.

The third contribution of this research is the technical infrastructure, which is open source. Anyone interested can download and use the system in study, teaching or professional development. Software ICU offers not only useful presentation of multiple software development metrics, but also highly configurable and customizable to satisfy various needs. Hackstat is only open source system that provides rich features of automated software engineering measurement and analysis and based on service-oriented architecture, which provide high extensibility. Users can easily configure, modify and/or extend the system according to their special requirements.

7.2 Future Direction

As discussed in previous section, Software ICU still requires more research and improvement to overcome its shortage.

First, vital signs' presentations require more research. The correlation between values of vital signs and the "health" state of a project is not yet concluded. Parameters need further tuning to

better indicate the “health” state of a vital sign. Eventually, more sophisticated vital signs may be needed as well as more sophisticated coloring methods to interpret them.

Second, it would be interesting to use the Software ICU in an industrial setting. The Software ICU offers powerful means to manage large number of ongoing projects and to fast interpret project “health” state, which meets the requirement in project management in industrial environment.

Last, but not the least, the completeness of DevTime data awaits further enhancement. The number of applications that are supported by automatic data collection sensors is too small, which may not only impair potential user’s motivation of using the system, but also lead to bias in different development activities. This shortage will probably raise the barrier to adopting Hackstat and Software ICU to industry and other software development environment. There are two ways to improve this. The first one is to implement more sensors to more development platform as well as more kinds of development applications. The second one is to provide user a way to manually report development activities, which may be difficult, if ever possible, to capture (e.g. reading paper-based material).

Appendix A

2008 Classroom Evaluation Questionnaire of Hackystat

Hackystat Evaluation

Hackystat is a long term research project concerned with improving the effectiveness and efficiency of software engineering metrics collection and analysis. Since 2003, we have periodically conducted a survey of students in ICS software engineering classes to assess the current strengths and weaknesses of the system.

To preserve anonymity, while also ensuring that only ICS students respond and respond only once, we ask you to provide the "secret code" that you randomly selected in class. To enable credit for completing this evaluation, only the graduate student researcher on this project (Shaoxuan Zhang) will know which code corresponds to you. He will provide a list of names who should be awarded credit to the class instructor without identifying individual responses. You can also contact Shaoxuan if you want your data deleted from analysis after you've submitted it.

If you want to go back and change your responses, simply fill out the entire form again. We will discard all but the most recently submitted entry for a given code.

This survey contains 17 questions and we expect that you will need about 10 minutes to complete it.

Thank you very much for your help! We take your views very seriously: prior responses to this survey have led to far-reaching changes in Hackystat.

Before filling out this questionnaire, you might want to take a look at the following image for the Software ICU to refresh your memory:

<http://csdl.ics.hawaii.edu/~johnson/portfolio.gif>

** Required*

*1. Installing the Eclipse IDE sensor was: **

- Very Easy*
- Easy*

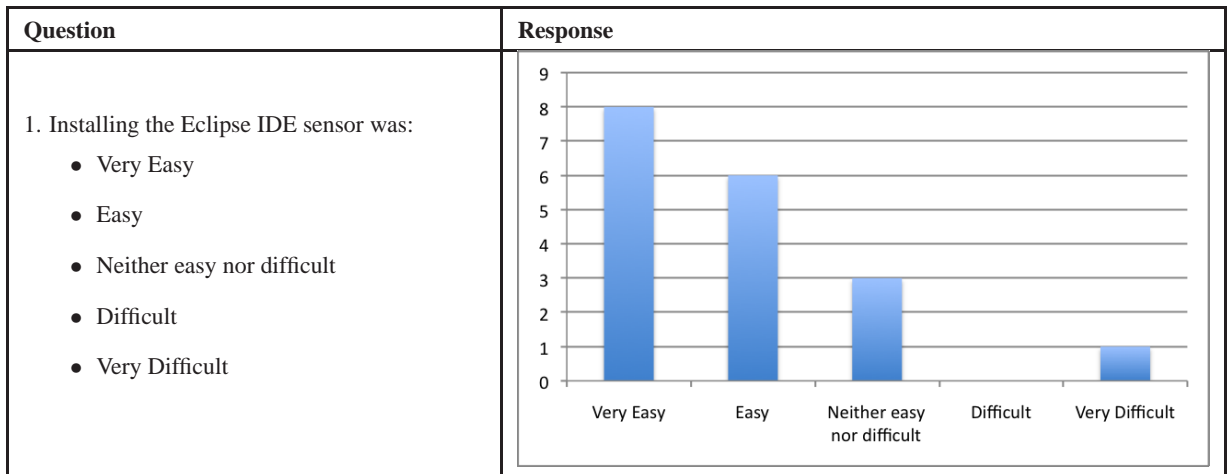
- *Neither easy nor difficult*
 - *Difficult*
 - *Very Difficult*
2. *Installing the Ant sensors (JUnit, SCLC, Emma, etc.) was: **
 - *Very Easy*
 - *Easy*
 - *Neither easy nor difficult*
 - *Difficult*
 - *Very Difficult*
 3. *Please provide any feedback you can on the problems you experienced during sensor installation and server configuration, as well as any suggestions you have to make this easier in future.*
 4. *The amount of overhead required to collect Hackystat data (after successful installation and configuration of sensors) was: **
 - *Very Low*
 - *Low*
 - *Neither low nor high*
 - *High*
 - *Very High*
 5. *The amount of overhead required to run Hackystat analyses was: **
 - *Very Low*
 - *Low*
 - *Neither low nor high*
 - *High*
 - *Very High*
 6. *Please provide any feedback you can on Hackystat overhead, as well as any suggestions you have to reduce the overhead in future.*
 7. *Did you encounter any problems while collecting data? Was there any kind of data that you failed to collect? If yes, please explain.*
 8. *How did you feel about sharing your software development data with other members of the class? **
 9. *How frequently did you use the telemetry page? **
 - *Every day or more*
 - *2-3 times a week*
 - *Once a week*
 - *Less than once a week*
 - *Never*
 10. *If you used the Telemetry page, what were you trying to find out?*
 11. *How frequently did you use the Software ICU? **

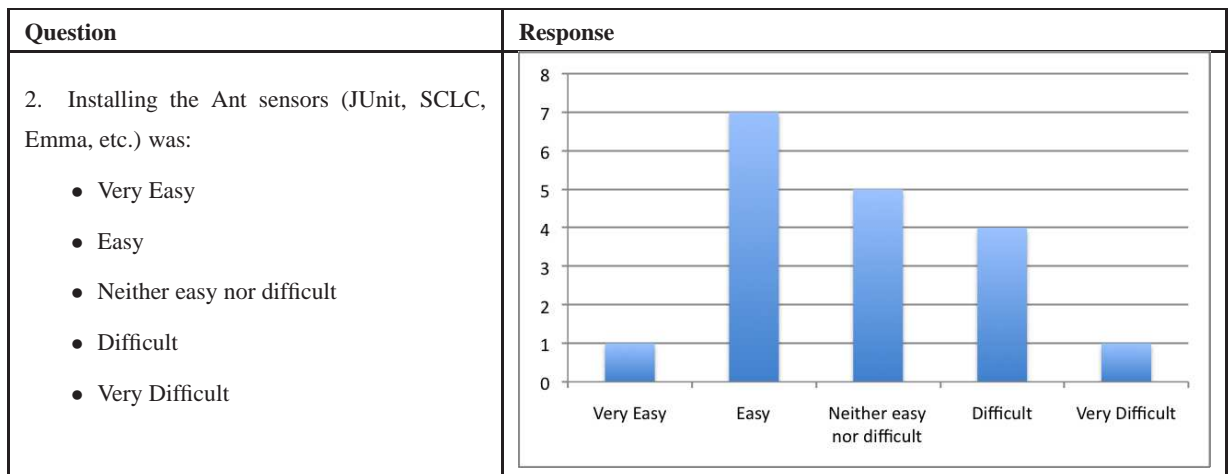
- *Every day or more*
 - *2-3 times a week*
 - *Once a week*
 - *Less than once a week*
 - *Never*
12. *If you used the Software ICU, please check the vital signs that were useful to you.*
*
- *Coverage*
 - *Complexity*
 - *Coupling*
 - *Churn*
 - *Size*
 - *DevTime*
 - *Commit*
 - *Build*
 - *Test*
 - *None of the above*
13. *Did you feel the Software ICU colors accurately reflected the "health" of your project? If not, why not? **
14. *Were you able to use the Software ICU to improve your software's quality and/or your team's process? If so, in what ways? If not, why not? **
15. *Please provide any other feedback you would like regarding Telemetry and the Software ICU, as well as any suggestions you have on how we can improve the system.*
16. *If I was a professional software developer, using Hackstat at my job would be: **
- *Very feasible*
 - *Somewhat feasible*
 - *Neither feasible nor infeasible*
 - *Somewhat infeasible*
 - *Very infeasible*
17. *Please provide any other feedback you can on the feasibility of Hackstat in a professional setting, as well as any suggestions you have on how its feasibility could be improved.*

Appendix B

Results form 2008 Classroom Evaluation Questionnaire of Hackystat

This section presents the responses from the respondents to each of the questions. For the “short answer” questions, I corrected misspellings and minor grammatical errors to improve readability.





3. Please provide any feedback you can on the problems you experienced during sensor installation and server conguration, as well as any suggestions you have to make this easier in future.

- I could not figure out what step makes a .hackystat directory. My .hackystat directory automatically generated in my Documents and Settings directory which has a blank space in directory name. I am still not sure how to move this folder to other. The installation of all sensors was pretty well described at the project homepage and there was no problems I have met during the installation.
- Both the installation and sending sensor data was easy. However, tracking down whenever there is a problem with the sensor is not so easy. A troubleshooting page in the near future?
- Installing the sensors was pretty straightforward. I didn't have any problems.
- Case sensitivity was one problem between user and Hackystat, but it was fixed.
If it is possible to have a .EXE that will automatically create environment variables and also install files into a local directory will be awesome.
- I did have one small hang up when installing the Ant sensors: If I remember correctly I was getting a NoClassDefinition error whenever a sensor ran. I was running java 1.5. I fixed it by downloading the jaxb libraries since the errors were referring to that. It could be not related to jaxb at all, but it worked after that. Otherwise, I had no problems whatsoever installing the sensors.
- Everything went smooth with the instructions given and the verification after each step.

- Personally I didn't run into any problems but some of the other students did. The sensors aren't difficult to install per se, but there are a lot of steps involved and it's easy to get lost while installing them. Maybe an automated installer can be created that searches for the Ant tools (maybe the user can provide a search directory) and will configure and install the sensors for the user.
- What made it hard was that all the instructions were not in one page. I had to go from one page to another and then to another. There should be instructions from STEP 1 to the end and provide proper links to the step by step process.
- First of all, the manual is too long. I do like your goal to analyze the software project, but if it wasn't required by this class, maybe I wouldn't think I want to use it, because it looks too complicated.

Also there are too many things that we need to download and install. If you want to encourage people to use this more, maybe you should provide a package of all the tools somehow.

For example, before it took a long time to install Apache, MySQL, PHP, and Perl, but now somebody offers a package called XAMPP, which is a combination of all of those, and entire installation finishes in 3 minutes. Something like that should be given.

- There is a lot of documentation in a lot of different places. It was confusing trying to figure out what to read in what order, and whether or not it was relevant to me.
- Some the installation instructions could benefit from "write once, use many times" as they're repetitive, which causes some people to start glossing over the instructions and then there's a couple that are slightly different and people (like me) won't notice the difference.
- The walkthrough was great, which made the installation easy.
- The only problem I had was the installation of the Ant sensor. I mean configuring it on Eclipse was easy especially when I try to run Emma, JUnit, FindBugs and all that from Eclipse it is sending stuff to Hackystat but when I checked my software ICU I didn't have any data on Build (all it says was N/A). And little did I know that when you run the ant sensors on Eclipse it only registers all the data to Hackystat JUnit, Emma, Checkstyle and such except BUILD. And I was told that running the BUILD on the command line works but not on Eclipse. So I tried that and YES that works. So is there a way to make it work on Eclipse when you run all the Ant sensors and it sends all the data to Hackystat including the BUILD data?

- When we ran the svn sensor, the build would fail if there are any commits from members not identified in our local Usermap.xml. Instead of looking for all commit records from all users within 24 hours, perhaps it could filter out and only look for records inside our UserMap.xml.
- The installation documentation must be read carefully. It may be easier to create a hackystat.build.xml with all the build targets, then import that file into each *.build.xml and call the sensor from the tasks.
- The most challenging sensor to get up and running was the SVN sensor. Other than that, the others seemed fairly easy to install.

Question	Response												
<p>4. The amount of overhead required to collect Hackystat data (after successful installation and configuration of sensors) was: *</p> <ul style="list-style-type: none"> • Very Low • Low • Neither low nor high • High • Very High 	<table border="1"> <thead> <tr> <th>Response Category</th> <th>Count</th> </tr> </thead> <tbody> <tr> <td>Very Low</td> <td>3</td> </tr> <tr> <td>Low</td> <td>7</td> </tr> <tr> <td>Neither low nor high</td> <td>7</td> </tr> <tr> <td>High</td> <td>1</td> </tr> <tr> <td>Very High</td> <td>0</td> </tr> </tbody> </table>	Response Category	Count	Very Low	3	Low	7	Neither low nor high	7	High	1	Very High	0
Response Category	Count												
Very Low	3												
Low	7												
Neither low nor high	7												
High	1												
Very High	0												
<p>5. The amount of overhead required to run Hackystat analyses was: *</p> <ul style="list-style-type: none"> • Very Low • Low • Neither low nor high • High • Very High 	<table border="1"> <thead> <tr> <th>Response Category</th> <th>Count</th> </tr> </thead> <tbody> <tr> <td>Very Low</td> <td>2</td> </tr> <tr> <td>Low</td> <td>5</td> </tr> <tr> <td>Neither low nor high</td> <td>9</td> </tr> <tr> <td>High</td> <td>2</td> </tr> <tr> <td>Very High</td> <td>0</td> </tr> </tbody> </table>	Response Category	Count	Very Low	2	Low	5	Neither low nor high	9	High	2	Very High	0
Response Category	Count												
Very Low	2												
Low	5												
Neither low nor high	9												
High	2												
Very High	0												

6. Please provide any feedback you can on Hackystat overhead, as well as any suggestions you have to reduce the overhead in future.

- Since the verify command runs all the tests, I'd think that it should send data for all tests run. Rather, in the portfolio analysis, the Unit Test portion only retrieved data for any JUnit builds

that were run. It doesn't really make sense why we'd have to run it separately when verify does it anyway.

- If I am correct, overhead - the processing time required by a device prior to the execution of a command. Then it all depends on what computer the user is using, I am using a single-core processor laptop it did not take long.
- Since Dr. Johnson provided us with Ant sensor examples, it was quite easy to set up everything to send data to the sensorbase. I did the hackystat tutorial and everything worked fine. However, I missed the part about creating a usermap.xml file for the svn sensors through Ant. That confused me a bit later on but I figured it out.

What made getting data quite easy as well was having Hudson installed on a dedicated continuous integration server. Daily builds would auto-send data to Hackystat and this made it super easy to get daily info.

- The sensors ran automatically and it was fast with sending the data.
- Maybe there can be a link on <http://dasha.ics.hawaii.edu> to both the Hudson and Hackystat server, that way we don't have to memorize the port numbers. Also, allowing us to create an account and password would go a long way towards usability. I had to put the Hackystat login information in a text file because I can't remember a randomly-generated string for the password.
- Sending sensor data was often quite slow. Generating reports in the web application was sometimes also slow – the page wouldn't load until you refreshed it.
- The overhead to collect data was generally small, however long enough that would generally run multiple (DOS) terminals so that I could continue working while it was sending data. Analysis was no overhead since that was just pulling up a browser page.
- When sending hackystat data, it was fairly quick on my computer, MacBook Pro. Tho, there were some students I saw which had a LONG wait time on the same laptop.
- I love Hackystat! It is a very great tool especially for a developer like me.
- Since Ant takes care of running Hackystat sensors, this made it very easy to accomplish.

7. Did you encounter any problems while collecting data? Was there any kind of data that you failed to collect? If yes, please explain.

- I had a problem with sending commit data to hackystat when I worked on a group project. That was because I did not update my sensors to newer version.
- At first during the implementation of DueDates 2.0, it was not collecting commit data from my account. It was due to the account on hackystat, it included the @gmail.com part of my gmail account. So it was not matching up with each other, the hackystat account and my gmail account.
- Running an analyses on my machine was slow, it would take over 3 minutes to run a build. I am not sure why it took so long to send the build data so I can't make a suggestion.
- Only JUnit data as mentioned previously.
- Case sensitivity was an issue at first, but it was corrected so I did not get problems after that. Hudson did not send to Hackystat number of commits, but that was fixed after a little modification with build.xml file.
- I was lucky. I rarely had any problems collecting data during all the time I worked with Hackystat. The one time something got screwed up was with my development time for one day. It said 0 when I checked and I had put in a bunch of time that day so it should have said otherwise.

I don't remember exactly but, that night I believe had worked in eclipse till after 12 at night, so it went to the next day before I closed the program. That could possibly be a reason for the missing data initially. The next day I just cleared the cache and it was all fine.
- There was a small issue when I first started collecting data, but it was quickly corrected when checking the xml files.
- Personally I ran into no problems collecting data.
- Sometimes it didn't collect build data for some reason.
- Occasional problems with SVN collection, I think, was a bit hard to tell.
- Everything was great except collecting data for my BUILD (please refer to above statement for more detailed problem regarding this). Thank you.
- I did with commit records but it was my fault. I wish subversion with Google Project Hosting would be more strict. I was able to check out the project with or without the "@gmail.com" suffix (i.e. "test" and "test@gmail.com"). Thus making me two different authors.

- Yes, the build data. I needed to set more environmental variables.
- For some unknown reason, my user name picked up the @gmail.com, so both my user name with and without @gmail.com needed to be added to the projects.

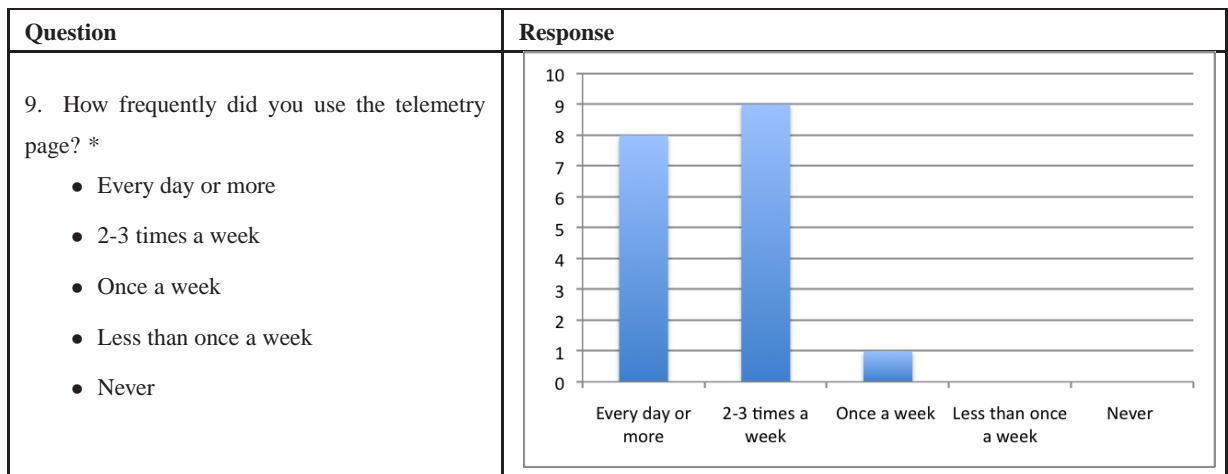
8. How did you feel about sharing your software development data with other members of the class?

- We could see how other groups were doing by sharing our software development data with other people. We also could find out what kinds of problems with our project by comparing graphs with other groups and this helped a lot.
- I was not offended if it was low, and I was quite intrigued with others data.
- I did not have a problem with sharing data with other people in class. I thought it was needed tool to keep tabs on everyone to assure they're doing their fair share.
- It felt good if your data was better than others. And if it wasn't, then you felt bad.
- Did not really like it because it is showing my programming habits, like starting on a project on the last couple of days.
- I felt alright about sharing my data with the class. It was interesting for me to see how other people worked on stuff. Some were consistent and others were not. Some people spend a lot of time working on stuff yet do not commit as much as others that work half the time. I think its good to see this data.
- I am okay with sharing my data.
- I didn't think it was a particularly good idea because it then forces group members to become competitive with each other, especially if one person is able to put in more time than all the others. Also, the data doesn't reflect the amount of work put in, maybe someone spent 5 hours doing research and only 1 hour programming, but the sensor data will only show 1 hour of development time and a minor code commit, versus someone who, say, just changes around the package structure for 3 hours and has a huge commit amount.
- Actually hackystat (or hacky-stalk as what my teammates and I called it) caused a lot of arguments and trash talk. Some guys were more concerned about collecting stats on hackystat than actually finishing the project. Some members would start competing on who had more

commits or move development time. The project turned out to be more of a competition of stats, which wasn't healthy for the team at all.

- It will be obvious that who worked on the project, so it is nice in terms of grading students. At the same time I feel some pressure that I need to work on the development, so if team leader require everybody to work well, this is good.
- Didn't really care.
- I had no problem with this, and it encouraged me to be aware of my time management and coding style.
- It was good in a sense that they can help you with test cases and coverage.
- It was fun..because you can see how everyone is doing within your group.
- Before taking this class, I didn't think that there was a way to track software development process. After learning about software continuous integration and working in a larger group project, I have a better insight in sharing the development process. I feel that it is a must in every software development environment, big or small to be able to communicate frequently and effectively.
- I was nervous because certain individual of the class seemed able to put in ridiculous long hours. I was concerned my amount of time (which seemed reasonable) would make me look as though I'm not working as hard.
- Good, I can see how I and others rank with each other.
- I am fine with this. All group projects in all schools (e.g., Architecture) should be required to use such a system. This is great for facilitating fair evaluations of students who participate, and those who 'get the grade' by riding on the laurels, blood, sweat, and tears of others.

Question	Response
----------	----------



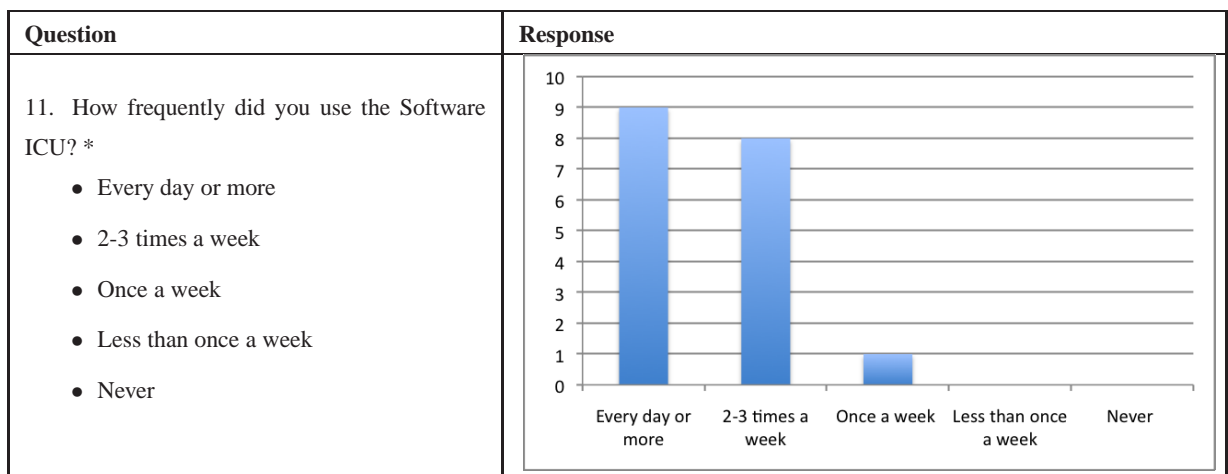
10. If you used the Telemetry page, what were you trying to find out?

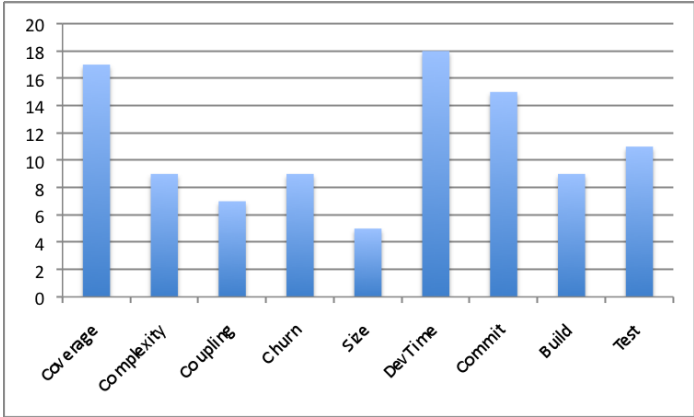
- I tried to find out how was I doing for the project by looking hackystat data.
- Seeing how much time i spent on the development of the program, and also others in my group.
- When I used the telemetry page I was trying to find out if I was on par with other groups members in terms of development, build, and commit numbers.
- Whether or not, my sensors were reading, and the work output of my group members (especially on days we didn't meet together).
- If my development time was up to par with my team members.
- I usually used the telemetry page to evaluate how my team was working overall, and what my part was in that data. I also checked it to make sure everyones data was being sent.
- It helps me see how I measure up with my partners.
- Member dev time mostly, to compare the amount of development time I put in vs. my group members.
- It supposed to show us how healthy individuals are in the group. So if one person is slacking, the members need to tell him to step it up. It wasn't used that way in our group. One person really wanted a good grade for the class so he just used the telemetry to watch himself; making sure no one gets more builds/devTime/commits than him (yes he said "i need more dev time

because i need an A”). I remember we had dinner as a group and one of our group members didn’t go to dinner. another group members then said “oh if he ups his stats more than mine, tomorrow I’m gonna hack all day.”

Sad, but true.

- member commit, member dev time
- Curious about trends in dev time, commits.
- Usually MemberDevTime, MemberBuilds, and MemberCommits. Basically just seeing how everyone was progressing.
- graphs, line trends of other group members
- My status and the status of our group and make sure everyone is doing their part.
- Mostly trends in individual performance, as well as overall project outlook.
- Basically if everyone was putting in the same amount of effort. Also it helped indicate if everyone is on track. If they have regular activity, then the chances of them on track is higher.
- Was the coverage, complexity and coupling getting bad?
- I tried to review each telemetry page daily to understand what I could do to improve the project health and focus efforts.



Question	Response																				
<p>12. If you used the Software ICU, please check the vital signs that were useful to you. *</p> <ul style="list-style-type: none"> • Coverage • Complexity • Coupling • Churn • Size • DevTime • Commit • Build • Test • None of the above 	 <table border="1"> <thead> <tr> <th>Vital Sign</th> <th>Frequency</th> </tr> </thead> <tbody> <tr> <td>coverage</td> <td>17</td> </tr> <tr> <td>complexity</td> <td>9</td> </tr> <tr> <td>coupling</td> <td>7</td> </tr> <tr> <td>churn</td> <td>9</td> </tr> <tr> <td>size</td> <td>5</td> </tr> <tr> <td>DevTime</td> <td>18</td> </tr> <tr> <td>Commit</td> <td>15</td> </tr> <tr> <td>Build</td> <td>9</td> </tr> <tr> <td>Test</td> <td>11</td> </tr> </tbody> </table>	Vital Sign	Frequency	coverage	17	complexity	9	coupling	7	churn	9	size	5	DevTime	18	Commit	15	Build	9	Test	11
Vital Sign	Frequency																				
coverage	17																				
complexity	9																				
coupling	7																				
churn	9																				
size	5																				
DevTime	18																				
Commit	15																				
Build	9																				
Test	11																				

13. Did you feel the Software ICU colors accurately reflected the health of your project?
If not, why not?

- I felt most of colors accurately reflected the health of the project. For the Coverage data, since we can write test cases just for increasing of the rates, we cannot assume that the project is in healthy condition even if the coverage data displayed in green color. However, I think this is not a problem of hackystat.
- Yes
- The only issue I had with the ICU colors was with the coupling. In both versions of DueDates we had to add extra classes at the last minute which would cause the coupling ICU to turn red. I am not sure how to address that because the coupling does need tracking.
- Not really, I don't think having a high churn amount is necessarily bad. Of course, it's a case-by-case thing. For my group, it wasn't about not committing frequently; we were just rehashing code because something just didn't work.
- Yes, reflected accurately on the health of the project. Showed how much coverage we had.
- I feel that the Software ICU did accurately reflect the health of my projects. For Due Dates 2.0, which was a longer project, the data was getting increasingly more meaningful as the

trends were over a larger period of time. It is good to look at things like devtime, commits, coupling, and coverage to see the color and the past trend because i think they really say something about the current state of the project.

To make it simpler, whenever I knew our project wasn't doing good and people weren't working regularly, the software ICU would have lots of reds and yellows. When I knew the project was doing better and people were working regularly, there were greens. It makes sense.

- The ICU was accurate with our project because it showed drastic spikes in all signs. This reflects our project in poor health.
- Not particularly because a project's health cannot easily be determined by just measuring numbers alone. For example, it's easy to increase coverage, but if a class has nothing but getters and setters and a toString method, does it really need to be tested? Of course not, but someone might feel compelled to do it in order to increase coverage and get a better health, but it's just a waste of time in my opinion. Also, DevTime is only measured from Eclipse but that doesn't measure things such as someone reading a book or looking up websites for information. It only measures active development in one program, forcing people to only use whatever IDE's Hackystat supports. The figures for complexity and coupling are hard to evaluate too. We want complexity to be low but sometimes it's unavoidable for it to be high, and should Hackystat show an absolute cut-off point where the complexity must be below a certain point for the project to be considered acceptable? Coupling is another one that falls under this category, if your program relies on a lot of outside libraries, can someone really determine an absolute value that the project's coupling must be under?
- Yes.
- maybe
- Coverage: perhaps too sensitive to drops/bounces in coverage. Churn: while you're working on a project, churn is going to vary, sometimes a lot. The trend colors were not helpful.
- Yes, I felt it was a relatively healthy project, and this generally showed, in the end. In the first half the colors reflected not as health of a project, which I'd agree as well. I'm not sure rising coupling was entirely a bad sign as things went along and functionality was added, as it was a slow steady rise.
- Sometimes. Hard to determine what will fall into green, red, or yellow.

- Yes definitely.
- It somewhat reflected the quality of our project. Maybe in some dark corner something is not thoroughly being depicted through the colors. Perhaps a suggestion is to use different color hues.
- Yes it was pretty accurately reflected.
- No, since I did not correctly configure the sensors.
- This is subjective... Usually the colors were spot on, however, they are quick to turn one way or the other depending on events that are being managed by the team (e.g., large code churns due to removal of unused code/imported code, etc.).

14. Were you able to use the Software ICU to improve your software's quality and/or your team's process? If so, in what ways? If not, why not?

- We can check how other members are doing for the project through the Software ICU and this helps a lot especially when we are working on the team project.
- Yes, for tracking if members were working on their tasks. Also how complex the program is increasing or decreasing.
- In my opinion, it is not clear if the ICU improved our system. Because other tools such as junit, findbugs, and pmd was easier to use to improve the application.
- If anything, keeping an eye on coverage helped us look out for what was being tested and what wasn't. Yes, showed how much coverage we had, and improve on that.
- I think for sure the Software ICU improves team process. More than just keeping people "in check" when grades are at stake, it provides an accurate way to assess what's being done and by whom. Our team got a lot out of checking up on the software ICU and assessing our team process. It seemed to get better over time.

As far as the software's quality, I think the Software ICU could be very useful in improving this. If my project for instance was in the red for complexity and coupling, and there were some code issues, I could see all this automatically through hackstat. Besides coverage stats though, my team did not really use the ICU to improve the software's quality.

- ICU was able to help us because it told us what needs to be focused or corrected.

- Personally, I only found Hudson useful because it's like running your code on someone else's computer to see if your environment is set up differently from a generic machine. I feel that the data for Hackystat is more something to look at out of curiosity rather than something to determine how well a project's status is because it's hard to base a project's health based on numbers alone and it might put unrealistic pressures on the team to make the project healthy for Hackystat when they can better spend their time developing instead.
- Yes.
- Yes, coverage tells me if we didn't write enough test cases.
- No. Coverage: already aware from Emma. DevTime, Commit, Build, Test: either team members did not look at the statistics, or they didn't care, because their habits did not change much. Others: not much we could do about the other statistics.
- Yes because able to manage our time and development fairly equally, and also notice spikes indicating bigger changes or problems.
- Yes, shows where we could improve as a group and improve as a programmer.
- Like in my case last time, I saw on Software ICU that I don't have a data on my BUILD. So because of that information I know what the problem is and it helped me to find a solution and figure everything out before it is too late.
- Our project ICU definitely described our lacking and late attempt to improve coverage. Due to the ICU, we were able to distinguish this fact quick and easy.
- The amount of activity helped us identify who was falling behind. Without offending our members by outrageously claiming their not working, we could tell by the sensors. Members can be more self-critical by looking at their individual data compared to the groups.
- Yes, by checking the coverage, complexity and coupling.
- Yes. By targeting coverage, dev time, coupling, and complexity, my team was able to improve all these into areas that were acceptable to us.

15. Please provide any other feedback you would like regarding Telemetry and the Software ICU, as well as any suggestions you have on how we can improve the system.

- I do not think the commits, builds, tests should be colored in because it all depends on how much the user does on the project. Is it possible to show line coverage instead of method coverage? The software ICU and telemetry was awesome tools in helping out with the project. It gave me visual stats on the project.

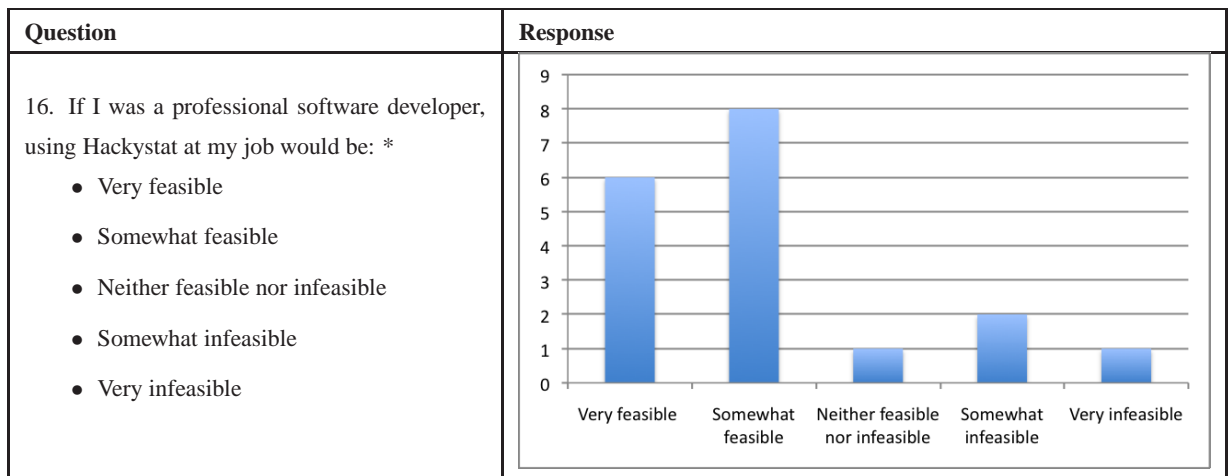
- What I think would be cool is to implement something to view the trend for each category in larger format but in the same style as the software ICU. I know this is shown on the telemetry page when you select it to show. However, I would be nice if there was some sort of rollover function that brought up a slightly larger window with a blown up overall trend. I can see how this isn't really needed but I would mostly likely check it a lot if it was there.

A minor thing that I noticed when using the Telemetry page was that when I selected a new statistic to view, the page would always jump back to the top and I'd have to scroll down each time. Its not really a biggie, but it makes navigating a bit slower when your going through all the project statistics.

- Consistent colors for each members can help.
- In addition to everything I mentioned above, it might help to somehow make the sensors configurable in some way, for example if two people are doing pair programming, there should be an option to set the sensors to send data for both people. Perhaps complexity can be measured somehow to only include methods that, say, start with get or set and toString. This way people aren't forced to write pointless test cases in order to increase coverage.
- Help page should be provided inside project browser. It should describe how to use it, what telemetry, what churn is, something like that.

Also your explanation should be simple so that people want to read it. If it is complicated and long explanation, nobody will read it.

- The different color bars and randomness might be fun and interesting, but I think having a bit more consistent scheme might be better. I would suggest if possible giving each developer a specific color that they always have during the project, either random, or chosen at the beginning.
- Does not capture development outside of Eclipse. For example, IMHO, MS Visual Studio is much better in the capacity as a web development IDE, which the dev time here was not recorded.



17. Please provide any other feedback you can on the feasibility of Hackstat in a professional setting, as well as any suggestions you have on how its feasibility could be improved.

- I think it's good to have this in a professional environment, cause the employer or client can check on how the progress of the program is going. With out having to make so much visits or hovering over workers.
- Cannot think of any off the top of my head. The Software ICU is already great for us programming students.
- I think Hackstat is definitely feasible in a professional setting, as long as it is supported in some way. For instance, if a team of developers is working on a project and they are all for having Hackstat manage project stats, that would be great. If, however, your the only person on your team that wants to use it, then it would be hard to send data that would assess team process.

I could see project managers wanting to have Hackstat data to evaluate everyone's input into the project, as well as the health of the project. Hackstat, I think, is perfect for new open source projects if releases are made early and often. It could be essential to seeing the overall health of the project.

- Overall, I feel like Hackstat would be an interesting tool to gather data to look at for curiosity's sake from time to time, but it should not be used as a basis for determining a project's health or to determine something such as member contribution. The sensors can only gather information from a few sources and these readings cannot account for a person's full contributions to a project. As for determining a project's health, I do not believe the sensor readings

can provide an accurate measurement because the sensors can only measure numbers based on algorithms, but it takes a person to really determine how good the code is.

- When I start to use hackystat, I need to get password from you and then eclipse send my data to your server. Some developers might have concern that hackystat steal source code.
- I think it depends a lot on the culture of job setting. I'm not too sure, but I think I may try setting it up on my own job site, even if just for myself to see my own trends.
- It is a very useful tool to keep track the health of a project so I would say it is feasible to have it in a job.
- My only wish is that ICU's should have a feature to support pair programming. Possibly a feature to indicate to the system that two people may be working on the same problem on the same system, rather than two individual machines. You might want to call this "collaborative mode", or something along the lines of that. These settings of course should be turned on or off easily from the developer's IDE (Eclipse).
- I work in a one person shop, so it would be difficult to say how useful this would be. As a lone developer, many metrics I am very cognizant of, however, having such a system would allow me to view those statistics that I do not have a "gut" feeling for. It would be great for my boss to measure the amount of time I spend on a project however.

Bibliography

- [1] James W. Moore Alain Abran. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.
- [2] T. DeMarco. *Controlling Software Projects*. Prentice Hall PTR, 1986.
- [3] Cem Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *10TH INTERNATIONAL SOFTWARE METRICS SYMPOSIUM*, 2004.
- [4] L. Buglione and A. Abran. Multidimensionality in software performance measurement: the qest/lime models. In *SSGRR2001 - 2nd International Conference in Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2001.
- [5] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Christopher Chan, Carleton A. Moore, Jitender Miglani, Shenyan Zhen, and William E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.
- [6] Watts S. Humphery. *A Discipline For Software Engineering*. Addison-Wesley, New York, 1995.
- [7] Watts S. Humphery. *Introduction to the Teasm Software Process*. Addison-Wesley, New York, 2000.
- [8] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley, 1995.
- [9] Alberto Sillitti, Andrea Janes, Giancarlo Succi, and Tullio Vernazza. Collecting, integrating and analyzing software metrics and personal software process data. In *Proceedings of the 29th Conference on EUROMICRO*, page 336. IEEE Computer Society, 2003.

- [10] Irina Diana Coman, Alberto Sillitti, and Giancarlo Succi. A case-study on using an automated in-process software engineering measurement and analysis system in an industrial environment. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 89–99. IEEE Computer Society, 2009.
- [11] Philip M. Johnson. Results from the 2003 classroom evaluation of Hackystat-UH. Technical Report CSDL-03-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, December 2003.
- [12] Philip M. Johnson. Results from the 2006 classroom evaluation of Hackystat-UH. Technical Report CSDL-07-02, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, December 2006.
- [13] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, Univeristy of California, Irvine, 2000.
- [14] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
- [15] Robert D. Austin. *Measuring and Managing Performance in Organizations*. Dorset House Publishing, 1996.