Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH

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

Abstract

Measurement definition, collection, and analysis is an essential component of high quality software engineering practice, and is thus an essential component of the software engineering curriculum. However, providing students with practical experience with measurement in a classroom setting can be so time-consuming and intrusive that it's counter-productive—teaching students that software measurement is "impractical" for many software development contexts. In this research, we designed and evaluated a very low-overhead approach to measurement collection and analysis using the Hackystat system with special features for classroom use. We deployed this system in two software engineering classes at the University of Hawaii during Fall, 2003, and collected quantitative and qualitative data to evaluate the effectiveness of the approach. Results indicate that the approach represents substantial progress toward practical, automated metrics collection and analysis, though issues relating to the complexity of installation and privacy of user data remain.

1. Introduction

An important goal for the education of modern software engineering professionals is the ability to appropriately collect, analyze, and interpret software development process and product measurements [1]. One approach to measurement education involves the reading and analysis of published case studies of measurement programs (such as [5]) as well as general texts on how to define, implement, and interpret software metrics (such as [6]). An advantage of such literature-centered approaches is the breadth of contexts and approaches that students can consider.

A second approach is experiential, in which students actually collect and analyze software metrics while performing software development. The best known experiential approach is the Personal (and Team) Software Processes [7, 8]. An advantage of such experiential-centered approaches is the acquisition of concrete skills in measurement collection and analysis by students. However, experience with the PSP and TSP has revealed that they can introduce substantial overhead into the software development process [3, 11]. This can have the unfortunate, unintended consequence of "teaching" students that software process and product measurement is impractical for "real" software development unless the organization devotes substantial resources (such as a separate software engineering process group) to metric definition, collection, and analysis.

Since 2001, we have been developing an approach to software engineering measurement called Hackystat. The goal of Hackystat is to provide useful process and product measurement collection and analysis facilities to developers without adding overhead to their daily activities. To accomplish this, Hackystat provides a set of "sensors" that developers attach to their development tools, such as their editor, build tool, test framework, and configuration management system. These sensors unobtrusively monitor development activities and send process and product data to a centralized web service. Developers can log in to the website to see the collected raw data and run analyses that integrate and abstract the raw data streams to support higher level interpretations. Developers can also configure "alerts" that watch for specified conditions in the data stream and send the developer email when these conditions occur. To support the varying product and process measurement needs of different development contexts, Hackystat has a modular architecture supporting the definition of custom "configurations". For example, we designed one configuration, called Hackystat-JPL, to support analysis of build process data for the Mission Data System project at Jet Propulsion Laboratory [9]. We also developed a configuration called Hackystat-UH to support measurement collection and analysis of student software engineering data.

In this paper, we present the results of a case study designed to assess the Hackystat-UH configuration. We collected quantitative and qualitative data designed to address four research questions. First, what level of overhead do student developers experience when using Hackystat-UH? Second, do the student developers find the analyses provided by Hackystat-UH to be usable and useful? Third, do student developers view Hackystat-UH as providing a reasonable approach to metric collection and analysis in "real world" settings? Fourth, how can Hackystat-UH be made more efficient and effective?

The next section describes the classroom context in which we performed the case study. We then introduce the measurement collection and analysis capabilities of the Hackystat-UH configuration, followed by the quantitative and qualitative results of the case study. We conclude with our lessons learned and future directions. This research contributes new insights on the design of software engineering metrics instruction, the strengths and limitations of this configuration of automated metrics collection and analysis, and the capabilities and potential pitfalls of these measures in an industrial context.

2. The Classroom Setting

We performed the case study during the Fall semester of 2003 in a senior-level undergraduate and an introductory graduate-level software engineering course at the University of Hawaii. The undergraduate course contained 10 students, randomly assigned to 3 project groups with three members each (one did not participate in the group project). The graduate course had 17 students, randomly assigned to 3 groups of three students and 2 groups of four students. Both courses followed the same basic curriculum, though the graduate section had additional supplementary readings.

The experiential curriculum focused on techniques and tools for Java-based software engineering of open source software systems. The techniques included: Java code formatting and documentation best practices, system packaging for multi-platform open source distribution, design patterns, team structure and organization, unit testing, coverage-based test plan assessment, configuration management, software review, web applications, and agile software development. The tools included: Eclipse (an interactive development environment), CVS (a configuration management system), Ant (a Java-based build tool), JUnit (an automated testing framework), HttpUnit (a testing framework for web applications), JBlanket (a coverage tool), Tomcat (a web application server), and Hackystat (our tool for process and product measurement).

For the first half of the course, the students worked alone to develop a series of small software systems to introduce them to the tools and techniques. In the second half of the course, the instructor divided the students into eight teams, each charged with developing a web application called "SiteWatch". The SiteWatch system is a JSP and Servlet-based system that enables a community of users to register with the installed web application and use it to manage their own personal "watch list" of web sites. The SiteWatch system recursively traverses the web sites on the watch list of each user each day, informing the registered user (via email or via a web interface) of any additions, deletions, or modifications to pages within the sites on their watch list.

The eight teams followed many of the practices associated with "agile" development. The requirements for the SiteWatch system were expressed as a set of 20 "user stories", short descriptions of end-user functionality. The professor acted as an "on-site" customer, providing feedback, resolving requirement ambiguities, and prioritizing user stories. The systems were developed incrementally over two months, with release iterations occurring approximately weekly. For a user story to be considered complete, the team was required to provide an automated acceptance test using JUnit and HttpUnit that verified its functionality. Using Ant and CVS, the teams were able to integrate, build, and test their systems one or more times per day. Although the curriculum included descriptions and short-term use of pair programming and test-driven development, these two agile practices were not required of the SiteWatch teams. Teams did see high-level descriptions of each other's systems at weekly review sessions, but were not allowed to download code from each other's CVS repository.

Unlike agile practices, which generally eschew detailed metrics collection and analysis, the students installed Hackystat sensors into each installation of their interactive development environment (Eclipse) and their build tool (Ant). These sensors collected information about the time each project member spent editing each file in their system, structural metrics of the system at the time of a build, the occurrence of JUnit test invocation and the unit test results, and the coverage associated with the unit tests. Students ran analyses over this data to learn about their own development process, to compare aspects of their development to other team members, and to compare their team's progress to that of other teams. The next section discusses these features of the Hackystat-UH configuration in more detail.

3. The Hackystat-UH configuration

Since its inception in 2001, a major goal of the Hackystat project has been to learn how to make existing software development environments "metrics-aware", freeing the developer from the burden of metrics collection and analysis so that they can focus on higher-level issues of interpretation and process change. We realized from the start that there could be no "one size fits all" approach: different organizations would require their own unique combination of sensors, metrics, and analyses. The current Hackystat system consists of over a dozen subsystems that can be composed together in different configurations depending upon the types of measures, sensors, and analyses desired by an organization.

We designed the Hackystat-UH configuration for use in the Fall, 2003 software engineering classes at the University of Hawaii. The configuration provides sensors for Eclipse, JUnit, BCML, and JBlanket, and assumes that all developers will use these tools. It also provides analyses based upon the assumption that developers will work in a set of groups, and that developers can gain insight into software development through access to measures regarding their own group's processes and products, as well as through access to measures that compare their own group to others in the class. The next sections present examples of these measures and analyses in the Hackystat-UH configuration.

3.1. Measures: Active Time and Most Active File

Software developer effort is a notoriously difficult metric to measure. At one extreme, the Personal Software Process requires developers keep a log of time spent in each phase of development, including design, implementation, compilation, defect removal, and so forth. Developers are advised to keep a stopwatch beside their desk, and use it to track the duration of each interruption during development. While this approach has (in theory) the potential to produce very fine-grained measures of effort, experience has shown that it is (in practice) both inaccurate and impractical: few if any developers willingly accomodate such measurement overhead for extended periods of time.

At the other extreme, industrial metrics programs often rely on timecard data to measure developer effort. This approach has the advantage of incurring virtually no additional overhead beyond already existing administrative tasks. However, the problem with this approach is that timecard data is very coarse in nature; it rarely yields information more specific than that the developer was engaged in some kind of activity related to the project. Even worse, timecard data is often inaccurate: a common administrative constraint is that developers cannot charge more than 40 hours a week to their projects regardless of how many actual hours they work.

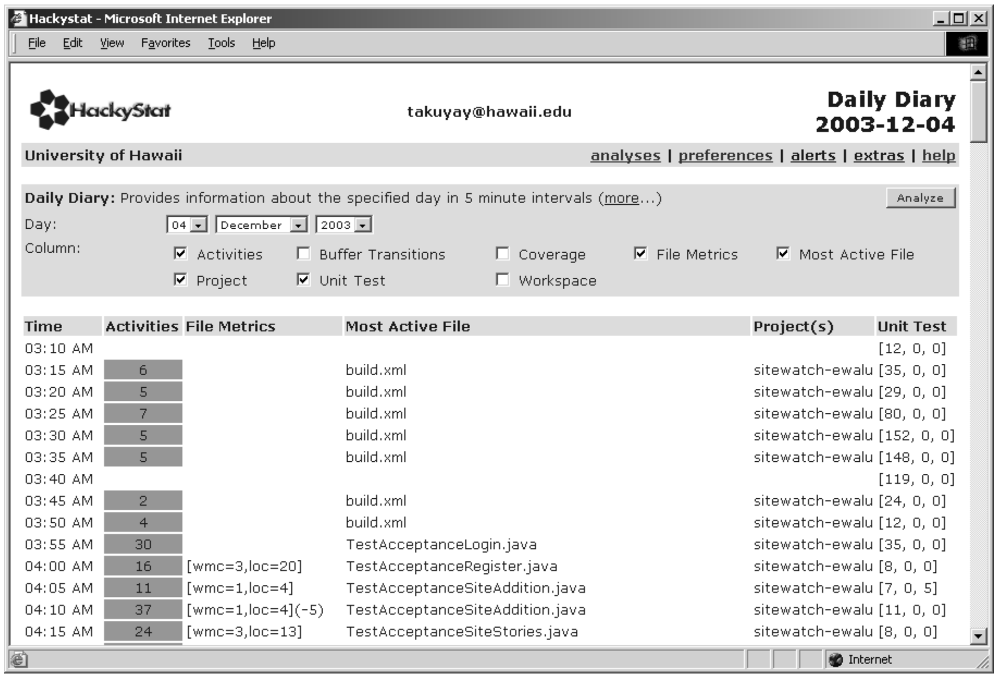
Hackystat takes an alternative approach to the problem of measuring developer effort, one that combines the fine-grained approach of PSP style measurement with the low overhead of timecard-based measurement. The cost of this precise yet low overhead effort measurement is completeness: Hackystat only measures the subset of developer effort that can be sensed from active use of development environment tools. To minimize confusion, Hackystat calls this measure as "Active Time" rather than "Effort".

We have developed Active Time sensors for several interactive development environments, including JBuilder, Eclipse, Emacs, VisualStudio, and Vim, and implementations for Microsoft Office tools are forthcoming. Each sensor implements the following algorithm. First, the sensor starts a timer-based process that wakes up at user-configurable intervals (with a default of 30 seconds). Each time the Active Time sensor timer wakes up, it determines the buffer (and file) the user is currently working on, and its current size. It then compares this information to the file and file size from the previous 30 seconds, and if the user is still working on the same file and if the size has changed, then the sensor records a "State Change" event for this file for this 30 second interval.

The Active Time sensor collects together State Change events and sends them to the server at user-configurable intervals (with a default of 10 minutes). The server processes State Change events to generate Active Time values in the following way. First, each 24 hour day is divided up into 288 five minute intervals. For each five minute interval, if at least one State Change event exists for that interval, then the user is assumed to have been active for that entire five minute interval. To determine the focus of attention during that five minute interval, the file associated with each State Change event counts as a "vote", and the file getting the most votes during that five minute interval becomes the "Most Active File" for that five minute interval, and is counted as the single file that the user was working on for that entire five minute period.

Figure 1 shows a portion of one user's Daily Diary, which is an analysis that illustrates (among other things) the results of processing State Change events into Active Times and Most Active Files. Although this approach of using a five minute grain size might appear to have the potential to lose information, we performed a validation study that indicated very little difference in results for a five minute grain size and, for example, a one minute grain size [12].

The Active Time/Most Active File measure has several appealing properties. First, it can be performed entirely automatically and does not incur any developer overhead. Second, the information enables a variety of higher-level analyses: for example, editing a file with the suffix ".java" implies that the developer was engaged in Java program de-


Hackystat - Microsoft Internet Explorer
File Edit View Favorites Tools Help
HackyStat
Analyze
Internet

takuyay@hawaii.edu

Daily Diary
2003-12-04

University of Hawaii analyses | preferences | alerts | extras | help

Daily Diary: Provides information about the specified day in 5 minute intervals (more...)

Day: 04 December 2003

Column: [✓ Activities] Buffer Transitions Coverage [✓ File Metrics] [✓ Most Active File] [✓ Project] [✓ Unit Test] Workspace

Time	Activities	File Metrics	Most Active File	Project(s)	Unit Test
03:10 AM					[12, 0, 0]
03:15 AM	6		build.xml	sitewatch-ewalu	[35, 0, 0]
03:20 AM	5		build.xml	sitewatch-ewalu	[29, 0, 0]
03:25 AM	7		build.xml	sitewatch-ewalu	[80, 0, 0]
03:30 AM	5		build.xml	sitewatch-ewalu	[152, 0, 0]
03:35 AM	5		build.xml	sitewatch-ewalu	[148, 0, 0]
03:40 AM					[119, 0, 0]
03:45 AM	2		build.xml	sitewatch-ewalu	[24, 0, 0]
03:50 AM	4		build.xml	sitewatch-ewalu	[12, 0, 0]
03:55 AM	30		TestAcceptanceLogin.java	sitewatch-ewalu	[35, 0, 0]
04:00 AM	16	[wmc=3,loc=20]	TestAcceptanceRegister.java	sitewatch-ewalu	[8, 0, 0]
04:05 AM	11	[wmc=1,loc=4]	TestAcceptanceSiteAddition.java	sitewatch-ewalu	[7, 0, 5]
04:10 AM	37	[wmc=1,loc=4](-5)	TestAcceptanceSiteAddition.java	sitewatch-ewalu	[11, 0, 0]
04:15 AM	24	[wmc=3,loc=13]	TestAcceptanceSiteStories.java	sitewatch-ewalu	[8, 0, 0]

Figure 1. The Daily Diary represents the user's day in 5 minute intervals.

velopment during that five minute interval. Third, it automatically senses idle time—if the developer goes off to lunch and leaves their editor running, the timer will stop recording State Change events within 60 seconds. Fourth, it is consistent and comparable: Active Time is measured the same for all developers regardless of the editor they are using. Fifth, it is specific and unambiguous: Active Time measures the time the developer spends physically modifying software artifacts.

It must be noted that not all important and useful developer activities involve modification of software artifacts. For example, design meetings can be extremely important and useful, but Active Time does not represent this effort. Managers may perform a critical role in improving project velocity by using email to coordinate people's activities, but Active Time does not represent this effort either. We will return to this issue in Section 5.

3.2. Measures: FileMetric, Coverage, UnitTest

The Active Time/Most Active File measure indicates the developer's focus of attention, but indicate little about the results of that attention. The Hackystat-UH configuration collects three additional measures that help track the evolution of the software system: FileMetric, UnitTest, and Coverage.

The FileMetric measure provides size and complexity data regarding the software artifact. Hackystat provides a sensor for the "BCML" (Byte Code Metrics Library) tool that calculates object oriented size and complexity metrics for Java based upon the Chidamber-Kemerer metrics definitions [4]. The BCML sensor is available for certain IDEs (such as Eclipse and Emacs) to calculate the current size and complexity for the Most Active File. An Ant task is also available for obtaining a snapshot of the system's overall size and complexity.

The UnitTest measure provides information regarding the invocation of JUnit tests and their results: whether they passed, failed, or generated an exception. As with the FileMetric measure, this data can either be collected within the IDE if a JUnit sensor is available, or through an Ant task. Currently the only IDE supporting a JUnit sensor is Eclipse.

Finally, the Coverage measure provides an indication of the percentage of the source code exercised by testing, and thus an indirect measure of the quality of testing. Hackystat provides a sensor for the JBlanket method coverage tool [2]. Both JBlanket, and the Hackystat sensor for collecting Coverage measures from it, are available to developers as Ant tasks.

As with Active Time, the FileMetric, UnitTest, and Coverage measures are designed to avoid incurring new overhead on developers apart from installation. The BCML sensor Ant task can be installed as a dependent target of the compile task, so that the structural metrics are automatically computed and sent to the server each time the system is built. Similarly, the UnitTest and Coverage sensors run each time the developer tests the system.

3.3. Definition: Projects

In addition to active time, file metrics, and test invocations and coverage, the Hackystat-UH configuration must represent which developers are working together, what files they are collaboratively developing, and the time period during which a given increment of the system is under development. While it might be possible for the system to attempt to infer this from the patterns of work of developers, it is more simple and less error-prone to simply require one developer from each project team to define a "Project" containing that information.

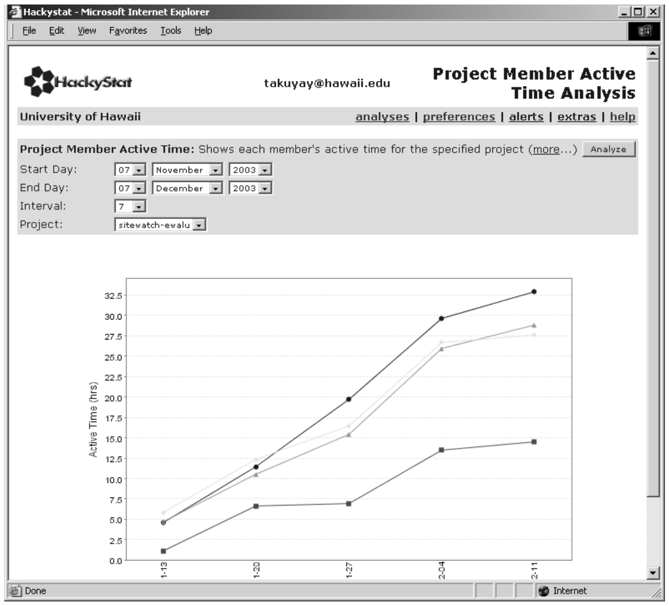
A Project definition specifies the set of developers who are collaborating together, as well as the directory hierarchies containing the project files. The system sends the developers specified in this project definition an email indicating that they have been defined within a project. They must explicitly confirm their agreement to participate in this Project definition, which implies their agreement to allow their data to be used in Project-level analyses that will be visible to all group members.

While defining a Project is overhead, it is a one-time overhead of only a few minutes per project. Since a typical Project lasts several weeks or months, this overhead appears to be acceptable provided developers receive benefits from the representation. The next sections present a subset of the analyses available in the Hackystat-UH configuration that provide a flavor for the kinds of information developers can obtain.

3.4. Analysis: Project Member Active Time

Two common problems in student project work are procrastination and free-loading. Procrastination leads to a delay in the start of the project and/or intervals where nothing is accomplished, followed by a last minute desperate rush to complete the system. Free-loading results in one or more members of the project team performing very little useful work, requiring excessive effort and commitment from the remaining team members to complete the project. Procrastination and free-loading tend to reduce the quality and completeness of the project, and increase the stress levels of participants.

Figure 2 illustrates the "Project Member Active Time" analysis. This analysis provides each project team with a chart of the cumulative Active Time devoted by each of their members to the project. Project Member Active Time is designed to ameliorate procrastination and free-loading in a group by making visible the consistency and equality of Active Time devoted to the project over time. If a team member is working consistently on the project, then their line will have an approximately constant slope. If team members are contributing an equal amount of active time, then



takuyay@hawaii.edu

Project Member Active Time Analysis

University of Hawaii

Project Member Active Time: Shows each member's active time for the specified project (more...)

Start Day: 07 November 2003

End Day: 07 December 2003

Interval: 7

Project: sitewatch-ewalu

Figure 2. Project Member Active Time

the each line should converge to the same final cumulative Active Time value. In Figure 2, for example, two of the three members generated an approximately equal amount of Active Time with a consistent slope. The third member generated less Active Time, and less consistently than the other two. The chart legend (omitted in Figure 2) indicates the team member associated with each line.

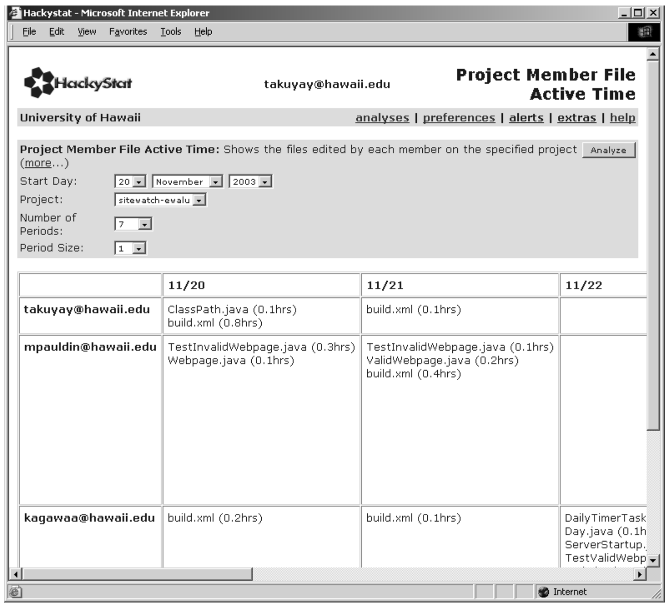
3.5. Analysis: Project Member File Active Time

A second problem in student project work is maintaining awareness of the work being done by members on a daily basis. Students tend to work in a highly distributed setting, with little time spent physically co-located. Lack of knowledge of who is working on what can lead to coordination problems and reduced productivity.

Figure 3 illustrates the "Project Member File Active Time" analysis. This analysis provides each team with a table showing the files associated with their project that a member of the team worked on for a given day, along with the number of minutes of Active Time associated with the file. The analysis provides a window into the daily development activities of each group member and the team as a whole. It is designed to provide each team member with a kind of passive "awareness" of the group's efforts.

3.6. Analysis: Course Project - To Date

The "Project" analyses provide information to members of a group about the process and products of that group's


Hackystat - Microsoft Internet Explorer
File Edit View Favorites Tools Help
HackyStat
Analyze
Internet

takuyay@hawaii.edu

Project Member File Active Time

University of Hawaii — analyses | preferences | alerts | extras | help

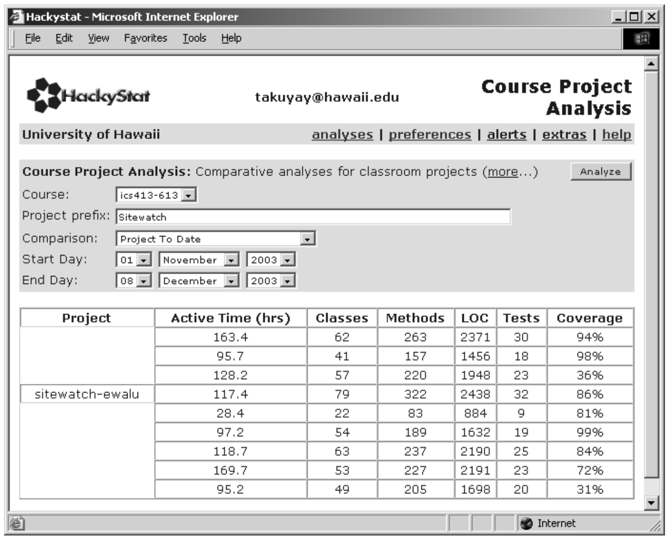
Project Member File Active Time: Shows the files edited by each member on the specified project (more...)

Start Day: 20 November 2003
Project: sitewatch-ewalu
Number of Periods: 7
Period Size: 1

	11/20	11/21	11/22
takuyay@hawaii.edu	ClassPath.java (0.1hrs) build.xml (0.8hrs)	build.xml (0.1hrs)	
mpauldin@hawaii.edu	TestInvalidWebpage.java (0.3hrs) Webpage.java (0.1hrs)	TestInvalidWebpage.java (0.1hrs) ValidWebpage.java (0.2hrs) build.xml (0.4hrs)	
kagawaa@hawaii.edu	build.xml (0.2hrs)	build.xml (0.1hrs)	DailyTimerTask Day.java (0.1h ServerStartup. TestValidWebp

Figure 3. Project Member File Active Time

software engineering efforts. However, there is also variability across student project groups: some get started more quickly, function more smoothly, and develop much high quality systems than others. The Hackystat-UH configuration provides a set of “Course” analyses that allow students to compare certain aggregate product and process measures across the set of all project groups. These analyses are designed to make visible the state and progress of all groups to each other, and thus help groups to recognize when and if they are “falling behind”.


Hackystat - Microsoft Internet Explorer
File Edit View Favorites Tools Help
HackyStat
Analyze
Internet

takuyay@hawaii.edu

Course Project Analysis

University of Hawaii — analyses | preferences | alerts | extras | help

Course Project Analysis: Comparative analyses for classroom projects (more...)

Course: ics413-613
Project prefix: Sitewatch
Comparison: Project To Date
Start Day: 01 November 2003
End Day: 08 December 2003

Project	Active Time (hrs)	Classes	Methods	LOC	Tests	Coverage
	163.4	62	263	2371	30	94%
	95.7	41	157	1456	18	98%
	128.2	57	220	1948	23	36%
sitewatch-ewalu	117.4	79	322	2438	32	86%
	28.4	22	83	884	9	81%
	97.2	54	189	1632	19	99%
	118.7	63	237	2190	25	84%
	169.7	53	227	2191	23	72%
	95.2	49	205	1698	20	31%

Figure 4. Course Project - To Date

The “To Date” analysis shown in Figure 4 collects together the latest values of the major measures associated with the Hackystat-UH configuration.

In this example, the SiteWatch-Ewalu system is the largest (in terms of classes, methods, and LOC) among all the SiteWatch projects, yet has lower coverage than several others. This indicates they might want to refocus development effort into increased testing.

3.7. Other measures and analyses

Space does not permit a description of all the analyses in the Hackystat-UH configuration. The Course Project Analysis provides six other analyses in addition to the “To Date” comparison described above, which allow teams to compare their process and product measures to each other over time. The configuration also includes a “Buffer Transitions” measure tracks the sequence of files visited by a user in their editor, which has potential to uncover “behavioral”, as opposed to structural, coupling between modules. A set of personal analyses on Active Time, files, and directory structures are provided. Hackystat provides a “alert” facility that users can configure in order to be notified by email when events of “interest” occur in their collected data.

4. Results and Evaluation

To assess the Hackystat-UH configuration, we collected both quantitative data focusing on the project outcomes and qualitative data focussing on the opinions of the students regarding Hackystat-UH after approximately six weeks of use.

4.1. Quantitative Results

The quantitative data includes the Hackystat measures collected throughout project development, along with the number of user stories completed by each project team. A summary of this data is provided in Figure 5. The completed user stories value provides an indirect measure of total project functionality, the total active time value provides an indirect measure of team effort, the final size provides an indirect measure of software complexity, and the final coverage measure provides an indirect measure of testing quality.

The completed user stories values indicate that all groups made substantial progress in development of the SiteWatch system, though only one group completed all 20 user stories in the project specification. All groups devoted substantial effort to their projects, with most groups spending between 95 to 130 hours. The systems varied in size between approximately 1500 and 2500 non-comment LOC. Coverage varied, with three groups obtaining good final coverage values (over 90%), two groups obtaining very poor final coverage (under 40%), and the remainder in the middle.

Perhaps the most interesting feature of the quantitative results in Figure 5 is that, despite substantial variation in the values of each measure across the eight teams, all of these measures are highly uncorrelated with each other. One might hope for a correlation between effort and functionality, for example, or between functionality and size, because such a correlation would support predictive measurement. At least for this context, any predictive model will require a more sophisticated representation of context, process, and product.

4.2. Qualitative Results

The quantitative data helps reveal the software development context in which we undertook a qualitative evaluation of the Hackystat-UH configuration. The evaluation consisted of a survey instrument containing 13 questions that we distributed via email to the 27 students in the two classes. Response was optional, but the students were offered extra credit points for providing their opinions. We obtained 24 responses, an 89% response rate.

The survey questions are organized into four sections. The Installation/Configuration section requests opinions regarding the ease or difficulty of initial installation and configuration of the Hackystat sensors and server-side configuration of the Hackystat user account. The Overhead of Use section requests opinions regarding the work required from users after installation and configuration to gather data and perform analyses. The Usability and Utility section requests opinions regarding the three analyses described in Section 3. "Usability" is defined to mean the ease of invoking an analysis and understanding what the results mean, while "utility" is defined to mean the usefulness of the analysis; does the analysis provide information that is actually helpful. The Future Use section requests opinions regarding whether Hackystat is considered feasible (i.e. appropriate, useful, or beneficial) for use in a professional software development context. Each section contains initial questions in which the subject was asked to respond with a number between 1 and 5 (1 being the most favorable response and 5 being the least favorable response, with their actual labels varying depending upon the question). The final question of each section asks for feedback on the section issue in an open-ended format.

A companion technical report to this paper provides online access to the complete questionnaire and the raw data [10]. Figure 6 presents a subset of the questions and the percentage of the 24 respondents replying with the given numeric value from 1 to 5.

The survey also includes four open-ended questions requesting feedback on problems encountered and suggestions for future improvements, which generated 98 comments (available in [10]). Example comments from each section include: "Yeah, a nice installation "wizard" with installation options and such would be kool", "After installation and configuration, there was virtually no overhead in using hackystat", "All of the above analyses were easy to use and understand. The Course Project Analysis was interesting to run to see how your group compared to the others, but I wasn't sure exactly how that information could be used to improve your project", and "I think Hackystat is very feasible in a professional setting. The only problem i can think of is that some people may not want to be monitored."

The quantitative and qualitative results provide evidence regarding the four research questions of this study.

1. What level of overhead do student developers experience when using Hackystat-UH? During installation and configuration, the level of overhead depends upon the specific sensor. The Eclipse sensor, for example, takes advantage of Eclipse platform installation wizards and thus incurs minimal overhead. The Ant sensors were significantly more difficult to install and thus incurred significantly more overhead. However, during daily use after successful installation, the results provide substantial evidence that Hackystat-UH is unintrusive and incurs very little overhead.

2. Do the student developers find the analyses provided by Hackystat-UH to be usable and useful? The results provide substantial evidence that Hackystat-UH analyses are quite usable—at least 75% of the respondents gave usability ratings of 1 or 2 to all three analyses evaluated, and no one rated them a 5 (Not Usable At All). Usefulness varied depending upon the analysis. Over 60% of the respondents gave the two Project Member analyses a usefulness rating of 1 or 2, and no one gave a rating of 5. However, the Course Project analyses received decidedly mixed reviews, with 50% giving them a rating of 1 or 2, but almost 30% giving them a 4 or 5.

3. Do student developers view Hackystat-UH as providing a reasonable approach to metric collection and analysis in "real world" settings? Over 70% responded with a 1 or 2 to the question of whether Hackystat would be feasible at their job if they were a professional software developer, and none responded with a 5. The open ended questions revealed concerns regarding privacy, although more than one response indicated that the respondent would not find the privacy issues of concern as long as they were the manager and not the programmer!

4. How can Hackystat-UH be made more efficient and effective? The results provide several priorities for future Hackystat-UH development. First, installation of Ant-based sensors should be simplified and made more reliable. One promising approach is a "download assistant" that automates the client-side installation process. Second, user doc-

	T1	T2	T3	T4	T5	T6	T7	T8
Completed User Stories	12	20	17	17	10	16	18	14
Total Active Time (hrs)	128	95	119	97	170	96	163	117
Final Size (LOC)	1948	1698	2190	1632	2191	2371	2438	1456
Final Coverage (%)	36	31	84	99	72	94	86	98

Figure 5. Selected quantitative results

Installation/Configuration	Very Easy		...		Very Difficult
Installing the Eclipse sensor was:	54%	33%	8%	4%	0%
Installing the Ant sensors (JUnit, JBlanket, BCML) were:	8%	17%	46%	21%	8%
Overhead of Use	Very Low		...		Very High
The amount of overhead required to collect Hackystat data was:	71%	13%	13%	0%	4%
The amount of overhead required to run Hackystat analyses was:	67%	17%	13%	4%	0%
Usability	Highly Usable		...		Not Usable at all
The Project Member Effort Analysis was:	54%	33%	8%	4%	0%
The Course Project Analysis was:	33%	42%	25%	0%	0%
Utility	Highly Useful		...		Not Useful at all
The Project Member Effort Analysis was:	38%	33%	13%	17%	0%
The Course Project Analysis was:	21%	29%	21%	17%	13%
Future Use	Very Feasible		...		Not Feasible at all
If I was a professional software developer, using Hackystat at my job would be:	38%	33%	25%	4%	0%

Figure 6. Selected qualitative results

umentation was weak in several areas. Third, the ability to track forms of developer activity in addition to Active Time would create a richer and more accurate representation. Two candidates include “Pair Programming Time” and “Code Review Time”. Finally, privacy issues are clearly an important aspect of Hackystat, and must be addressed in each proposed context of use.

4.3. Limitations

There are several threats to the external validity of this case study that must be taken into account in interpreting these results.

First, this data is drawn from a limited sample size of approximately two dozen students in software engineering classes at the University of Hawaii. 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.

Third, the administration of the questionnaire was performed by the designer of the system under study, who was also the instructor for the class. Responses were not provided anonymously, but rather emailed back to the instructor/designer. This raises the question of whether the responses are biased, either consciously or unconsciously, in order to ”please” the instructor/designer.

The best way to address each of these threats is to replicate this study in other academic and industrial environments, and we would support and encourage such efforts by interested independent researchers and practitioners.

5. Lessons Learned and Future Directions

Our first lesson is that the Hackystat-UH configuration does provide practical automated process and product metric collection and analysis in a classroom setting. This case study confirms and extends results from our previous research, in which we compared prior experiences with Hackystat to our experiences with the LEAP automated toolkit and with the Personal Software Process [11].

A second lesson is the need to learn more about the relationship between Hackystat’s representation of developer activity and overall developer effort. To that end, one future direction is to ask developers to manually keep paper-based logs of their total effort on a software project for short periods of several days, and then compare these logs to the sensor data to look for relationships between Active Time and overall developer effort. As noted above, another future direction is to develop additional representations for developer effort such as Review Time and Pair Programming Time.

A third lesson is the need to better understand privacy issues in automated measurement collection. The level of privacy in Hackystat depends upon the configuration: a configuration like Hackystat-JPL maintains complete developer-level privacy, while the Hackystat-UH configuration creates a level of transparency regarding activities that is novel to most developers. A final future direction is to investigate the organizational contexts that influence developer attitudes towards measurement privacy, and ways in which to make effective measurements without exceeding developer privacy comfort levels.

References

[1] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE, 2001.

[2] J. M. Agustin. Improving software quality through extreme coverage with JBlanket. M.S. Thesis CSDL-02-06, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, May 2003.

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

[4] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), June 1994.

[5] M. K. Daskalantonakis. A practical view of software measurement and implementation experiences within Motorola. *IEEE Transactions on Software Engineering*, November 1992.

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

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

[8] W. S. Humphrey. *Introduction to the Team Software Process*. Addison-Wesley, New York, 2000.

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

[10] P. M. Johnson. Results from qualitative evaluation of Hackystat-UH. Technical Report CSDL-03-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, December 2003. http://csdl.ics.hawaii.edu/techreports/03-13/03-13.html

[11] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.

[12] H. Kou and X. Xu. Most active file measurement in Hackystat. Technical Report CSDL-02-09, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, December 2002.