Measures for Mobile Users: an Architecture

Alberto Sillitti, Andrea Janes, Giancarlo Succi, and Tullio Vernazza

Abstract—Software measures are important to evaluate software properties like complexity, reusability, maintainability, effort required, etc. Collecting such data is difficult because of the lack of tools that perform acquisition automatically. It is not possible to implement a manual data collection because it is error prone and very time expensive. Moreover, developers often work in teams and sometimes in different places using laptops. These conditions require tools that collect data automatically, can work offline and merge data from different developers working in the same project. This paper presents PROM (PRO Metrics), a distributed Java based tool designed to collect automatically software measures. This tool uses a distributed architecture based on plug-ins, integrated in most popular development tools, and the SOAP communication protocol.

Index Terms—Development Monitoring, Process metrics, Product metrics, Java.

1 INTRODUCTION

It is difficult to extract measures from both software [7] and software development process [11] due to a shortage of focused tools that perform such tasks automatically. In particular, tracking the entire development process manually is time expensive and error prone [5], [14]. Moreover, these errors are more frequent when a reliable tracking is more important to understand if and how the process should be improved. For example, that happens when developers are under pressure because of an approaching deadline.

Metrics data are important to find out the correlation between objective measurable data and software qualities like complexity, reusability, cost of maintenance, effort required, etc.

A completely automated tool that performs such data acquisition without any effort by developers should help both developers and managers to improve software quality and shipping times. The system has to take care of developers’ privacy allowing managers and colleagues to access only aggregated data at different levels.

Collected data can help managers to implement a popular accounting technique called activity-base costing (ABC) [3]. Usually, this technique is difficult to implement in a software company because nearly all costs are human costs and keeping track of time spent in each activity is very difficult. The proposed system provides such data automatically.

This paper presents the architecture and the implementation of PROM (PRO Metrics), an automated and distributed Java tool for collecting and analyzing software metrics and personal software process (PSP) data. The paper is organized as follows: section 2 and subsections present an overview of the metrics collection problem; section 3 describes the state of the art; section 4 presents the architecture of PROM; section 5 describes the implementation; finally, section 6 draws the conclusions.

2 METRICS COLLECTION

2.1 Metrics Overview

Software metrics is an area that tries to quantify different aspects of software development. “You cannot control what you cannot measure” [4], this is the main reason to measure software. The goal of software metrics is finding a way to control software project that too often are over time and over budget.

Measuring is a process through which numbers or symbols are assigned to entities of the real world in order to describe it [7].

A measure is not only required to control, but also to verify, estimate, and help decision makers. Hardly a qualitative
approach is enough, without a quantitative approach is difficult to compare experiences, collect results, and perform analysis.

Through measures, engineers can evaluate a product that already exists or estimate features of a product that do not exist. In both cases, a model of the features of the artefact under analysis helps to identify relationships between artefact attributes. The COCOMO model [1] can be used in this sense. As instance, the equation \( E = aS^b \), if \( E \) and \( S \) are known, can be used to evaluate the position of the current project compared to the average from which the constants \( a \) and \( b \) are calculated; if \( S \) is estimate, the model can be used to estimate \( E \). A predictive system includes a model and a set of procedures to predict the unknown parameters of the model. The COCOMO model include a model described by the previous equation and a set of procedures to predict parameters \( a \), \( b \), and \( S \).

In an industrial context, software is characterized by:

1. Product quality
2. Process quality

A product is the result artifact of the software production; a process is the set of activities required to produce intermediate products and the final product. In the software context, the meaning of quality is not trivial. The IEEE Standard for Glossary of Software Engineering Terminology [12] defines quality as the whole set of features of a product or service able to satisfy a specific need.

Different people are interested in measure and control different variables. Managers are interested in:

- Cost: to determine the price of the product
- Productivity: to built a team with the right size
- Product quality: to compare different products
- Effectiveness of methodologies and tools: to choose the best ones

Developers are interested in:

- Product quality: to evaluate the status of a project
- Process quality: to improve it through ad-hoc changes

The analysis of the measure of several attributes determines an estimation of the quality of a software product [7]. These attributes include:

1. Internal attributes: features measurable through a static analysis of the product
2. External attributes: features requiring a comprehensive measurement of the product and its run-time behavior

External attributes such as reliability, performances, cost, etc. are difficult to measure. For this reason, they are often estimated using some, strictly related, internal attributes. The measurement of internal attributes during a project helps to identify quality problems in the final product.

Software metrics aim at quantifying properties of artifacts generated during the software lifecycle, of the development process, and of the associated resources [7].

The idea of industrializing the software development process is centered on the ability to identify and institutionalize advantageous software development practices. These “best practices” can be the cornerstone of the effective and efficient development of sound software systems.

Unfortunately, most of the proposed “best practices” still lack a sound, quantified assessment. Major problems include the identification of a substantial amount of relevant industrial data to analyze with a sound application of modern statistical techniques. Most of the studies refer to student data. The way students develop software cannot be considered representative of professional development. Statistical techniques are often used without a careful determination of their applicability.

Major theoretical and practical problems prevent the drawing of general conclusions applicable in different contexts even in the few cases when individual projects have evidenced the pros of a methodology or tool. In principle, generalization of findings coming from a particular study requires that we must make sure that the samples of developers, software, and the history of the software are randomly chosen from their respective global populations. This is seldom the case, since most of the individual studies in software engineering are performed for a specific software development environment.

### 2.2 Data Collection

In software engineering, measures are difficult to collect due to two main problems [5], [14]:

1. Collecting metrics is a time expensive task. This is a problem because software projects are often late and there is no time to spend in activities that do not produce immediate benefits;
2. Manual data collection is an unreliable activity. Too many errors or missing data badly affect the analysis process. These errors appear mostly in critical periods, when the data correctness should help to understand better the situation such as during high stress working periods.

Understanding how an enterprise really works is not an easy task and requires a huge amount of time spent interviewing both managers and employees. Managers of specific enterprise functions should know, in a very detailed way, how tasks are performed. Often, they know the big picture but their knowledge of the details is very different from how tasks are really performed. That could happen due to several reasons. Among them, there are the following:
There are no or limited quantity of data regarding how people work. People claim to work in a certain way but they actually work differently. The former is possible because inspections are very expensive, do not produce immediate benefits and too many errors or missing data badly affect the analysis process. These errors appear mostly in critical periods, when the data correctness should help to understand better the situation such as during high stress working periods. The latter is possible when management introduces wrong or too complex processes that people hardly follow but they claim to do. In this way, there is a mismatch between the real process and managers' knowledge. Unfortunately, it is difficult to collect useful data in software development environments because managers and developers do not consider measurement an important activity, compared to coding. Moreover, it is hard to collect data manually and there are only few, very expensive tools to collect the data semi-automatically. Semi-automatically means that still the human intervention is heavily involved, especially in the critical task of collecting process measures. Such tools suffer from severe limitations. (A) Mostly, they deal with product measures; (semi)automatic collection of process measures is nearly always ignored. (B) Often, they are not integrated in the “usual” working context of developers and managers; the developer is required to invoke such tools explicitly. This lack of integration affects the precision of the collected data. Sometimes, it even happens that measures are collected later in the process than expected, just to comply with given process guidelines; this results in spurious data. (C) Even when some of the tools a developer or a manager may use supports the collection of a few, mostly product-oriented measures, such measures are not automatically combined with the measures collected from other tools the user or the manager may be using. This still requires the developer and the manager to store explicitly the data with the consequent problems mentioned above; therefore, it limits the possibility of extracting sound, sensitive information from the combined analysis of measures. The lack of fully automated process and product measures collection tools affects severely the PSP and other similar approaches. Empirical investigations on the PSP evidence that measures data are usually not collected at all or with the required precision to produce meaningful results [13]. Automated data collection, integrated in popular software, helps to solve both problems and introduce further benefits such as help to implement the Activity-Based Costing (ABC) as accounting technique [3]. These automated data collection is possible wherever people spend most of their time performing computer-based activities such as software development. Engineers need to measure relevant variables of a process to understand and control it. That also happens in software engineering. The main resource in many companies, such as software companies, is human resource. For this reason, software engineers are mainly interested in: human effort needed to complete a task, quality, cost, and time required. Process metrics describe process qualities such as effort required, production time, steps of a task, etc. These qualities can be evaluated through the acquisition of measurable properties such as editing time, number and type of changes in a file, usage patterns, etc. Most of management costs are human resources costs: experience, skills, etc. Moreover, productivity of very good employee is tens times better than average [20]. For these reasons, it is very important to understand how top developers work and force all to adopt a process that helps them to achieve the best results possible. The Personal Software Process (PSP) [11] is a process for development improvement on a personal level. It is a self-improvement process to control, manage, and improve one's work. There is a structured framework of forms, guidelines, and procedures for software development. PSP uses the idea of feedback to achieve improvement. Developers define, measure, and track metrics such as code size, defects, effort, defect-removals, and so on. They evaluate and learn based on their defined processes and measurements. The measurements help to improve a developer's ability to estimate code size and effort before starting. They also aid the developer in becoming more efficient in doing reviews. Humphrey stresses the use of checklists to aid in design and code reviews. A checklist makes the review process more complete, formalized, and efficient. As new types of defects are found in the development process, they are appended to the checklist. The checklist moves the effort of fixing defects up to an earlier phase, where the repair is magnitudes less costly. The greatest goal of the PSP is for developers to discover the methods and practices that work for their abilities and adapt the process for their own use. Humphrey claims that improvement on the personal level ultimately makes developers better team members. PSP include several phases to help developers to trace a data set that includes, among others, working times and defects. At the end of each phase, collected data are ordered, related to further information such as code metrics, and stored in ad-hoc forms (plan summary forms) in a Postmortem Phase. At the end of all phases, developers update their private database of historical information regarding the productivity. PSP and several other research initiatives evidence the importance of collecting such data in software engineering and cross-analyzing them to have a benchmark for improvement. In addition, Extreme Programming and the other agile methodologies rooted in lean management inherit from lean management the critical importance of measuring [16]. PSP requires the collection of detailed metrics of the development time, bugs discovered and corrected at all development stages, and software size. Then all collected data are analyzed through statistical methods. These results provide software engineers sets of historical data mainly used to:
1. make reliable estimates on variables such as time schedule, quality, etc. of ongoing projects
2. find out how to improve the development process identifying problems

PSP requires the collection of the following data:
• Editing time: time spent performing a task such as coding, writing a document, etc.
• Tool name: the name of the tool used
• File name: the name of the file edited
• Class name: the name of the class edited (if possible)
• Project: the name of the project to which the document belongs
• Size: total size of the document
• Differential size: size of the document compared to the previous version
• Defects: defects identified in the source code

Due to the amount of information required, manual data collection is time expensive and error prone. An automated data collection is the only way to collect reliable data without affecting the process under analysis.

2.3 Metrics Correlation

Code and process metrics measures different but related attributes. Code metrics are able to identify potential problems inside the source code but they do not provide any data regarding how they have been generated. On the contrary, process metrics trace the behavior of developers but they do not collect information regarding the modification included in the source code.

A comprehensive approach is the best way to collect data regarding how developers spend their time and the effects of this effort on the source code. As a result, a huge effort spent in a single class can produce a remarkable improvement of the code or not. The only way to collect this information is through the comparison of code and process metrics data.

3 STATE OF THE ART

At present, there are several tools to address the metrics collection problem. Some of them are commercial products such as MetricCenter [15] or ProjectConsole [17]; others are research tools such as Hackystat [10] developed at the University of Hawaii. Often, these tools use proprietary solutions, domain dependent, or focusing on a specific aspect of data collection or analysis.

The basic idea of the Hackystat system is very close to PROM. Both are based on open source software and use plug-ins to collect data, but the aim of the two tools is very different. Hackystat focuses on the coding activity and target users are developers; PROM focuses on the entire development process including both code activities and non-code activities such as writing documentation, project management, etc. The target users include all members of the development team (including developers, managers, etc.).

In addition, PROM offers both detailed data for developers and high-level views, at project level, for managers helping them to monitor and improve the whole development process.

One of the goals of the PROM system is the collection of data and the retrieval of important information through a wide range of devices, including laptops and PDAs. The system is able to collect data even if the user is working offline using a synchronization mechanism when that system is online again.

Table 1 briefly compares the main features of the two systems.

4 DISTRIBUTED ARCHITECTURE

PROM (PRO Metrics) is a distributed Java tool designed to collect different set of software data: software metrics [7] and Personal Software Process data [11]. The former set includes code length, inter-class and inter-function dependencies, reusability, etc. The latter includes time spent in each activity, number of changes per class, etc.

To collect such data, the architecture has three main components (Fig. 1):
1. PROM core: this is a Java based architecture that includes three main components: a database that stores all data, a centralized PROM Server, and many distributed Plug-ins Servers (one per client). It provides web administration tools and access to collected data through dynamically generated statistics.
2. tool-specific plug-ins: they listen to events generated by third-parties tools and send relevant data to the PROM core [19]. Plug-ins are tool-dependent, for this reason they are written using different languages. A present, in available or under development plug-ins, these languages are: Java for NetBeans, Eclipse, Borland JBuilder, and Together Central; C++ for Microsoft Visual Studio 6.0, C# for Microsoft Office XP and Visual Studio.NET.
3. third-parties tools: the system uses popular tools for collecting data. Such tools include IDEs (Microsoft Visual Studio, NetBeans, Eclipse, Borland JBuilder, etc.), design tools (Rational Rose, Together Central, etc.), and office automation tools (Microsoft Office, Sun Staroffice, Open Office, etc.).

The Hackystat system developed at University of Hawaii [13] partially inspired this tool but the two systems have different targets: the former provides the most benefits to developers focusing on PSP; the latter has a wider target, it provides information to both managers and developers using different views on collected data that include software
The architecture of PROM fulfills these main constraints:
1. it is extensible to support new third-parties tools
2. the development of plug-ins for third-parties tools are as simple as possible
3. it supports both online and offline users
4. it provides data to both managers and developers preserving privacy
5. it supports different kind of clients (desktop, laptops, and handler)
6. all collected data are accessible through the SOAP protocol [18]

To fulfill such requirements the architecture is presented from different points of view. Fig. 1 describes the architecture of PROM from the point of view of data collection fulfilling the first constraint. Fig. 2 focuses on the same architecture as a layered structure satisfying constraints 4, 5, and 6. From this point of view, the architecture includes three layers: a database (layer 1), a server (layer 2), and a set of clients (layer 3).

The PROM Database (layer 1) stores all collected data, results of analysis, project details, and data regarding users. The PROM Server (layer 2) provides access to all information stored inside the database and make them available to clients in very different ways. Users can access data according to their role (developers or managers). Developers can access their own data; managers can access only aggregated data at different levels to preserve privacy of developers. Moreover, users can allow colleagues to access their own data. Clients (layer 3) include different kinds of devices: desktops, laptops, and PDAs. The first two set of devices can fully access to the system and collect data (software metrics and PSP data) through plug-ins attached to third-parties tools; PDAs cannot collect such data but they can provide information to managers even if they are mobile users accessing reports and statistical data important for the decision process.

Devices that collect data (desktops and laptops) require a set of three entities to satisfy requirements 2 and 3 (Fig. 3): a third-party tool, a plug-in, and a Plug-ins Server.

Developers use third-parties tools to design, write software or documentation. Tool-specific plug-ins listen to events generated inside those tools and collect relevant information. A Plug-ins Server collects such data and sends them to the PROM Server. Each client collecting data needs a Plug-ins Server to simplify plug-ins development. Plug-ins are tool dependant, so a specific implementation is required for each supported tool, while the Plug-in Server is written once because it is not tool dependant. For this reason plug-ins delegate as much as possible to the Plug-ins Server. This server provides a set of services to plug-ins:
1. users authentication
2. data caching to support offline users
3. data storing into the PROM Database through the PROM Server

System administration and reports are available through dynamically generated web pages that an application server provides retrieving data through the PROM Server (Fig. 4).

System administration includes: users and projects management (add, remove, update, assign users to projects, etc.), event collection parameters (types of events, frequency of collection, etc.), system status reports, etc.

The PROM system provides many different views on data. Detailed ones are designed for developers; aggregated views at project level are useful for managers. A deeper analysis is possible integrating into the system a specific analysis tool that can access data through the SOAP protocol and perform custom analysis.

The architecture also includes data analysis and visualization (Fig. 4). The PROM Server performs simple data analysis retrieving data from the database and aggregating them. The analysis considers also code evolution keeping track of versions of the source code stored into a CVS repository. When a piece of code is stored into a version control system, the Plug-in Server invokes the metrics extractor (WebMetrics) that collects metrics data and stores them into the PROM Database. This feature provides a comprehensive view of the development process to developers.

Data visualization is performed in two ways:
1. through web pages: a user can retrieve information using customizable but pre-defined queries. The user select a specific query, such as total time spent in a project, than inserts some parameters such as time constraints, and the system returns a web page with the required data. Among these pre-defined elaborations there are: effort spent in a project by users, effort spend by a user considering all projects, metrics-specific graphs (i.e. length of code, complexity, coupling, etc.)
2. using third-parties tools: in this case the system provides all the data to a specific tool that can perform user-defined analysis

4.1 WebMetrics
WebMetrics is an extensible tool able to extract code metrics from source code files. At present, implementation of the C/C++ and Java parsers are available.

This tool is able to extract the following metrics [7]:
• Procedural code metrics
• Object-Oriented design metrics

Procedural code metrics measure internal attributes of functions and procedures, so they are applicable to almost all programming languages. In particular, the following metrics are considered:
- Lines Of Code (LOC)
- McCabe Cyclomatic Complexity
- Halstead Volume
- Fan-In and Fan-Out

In addition, metrics have also been proposed for designs, especially object-oriented (OO) designs. Since OO designs can be well-defined and specified, as with a language like the Unified Modeling Language (UML), measures about a system’s class structure, coupling, and cohesion can be easily derived.

The most well-cited OO metrics are the Chidamber and Kemerer (CK) suite of OO design metrics [2]. This is a set of six metrics which capture different aspects of an OO design, including complexity, coupling, and cohesion. These metrics were the first attempt at being OO metrics with a strong theoretical basis. The metrics are listed below:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number Of Children (NOC)
- Coupling Between Object Classes (CBO)
- Response For a Class (RFC)
- Lack Of Cohesion in Methods (LCOM)

WebMetrics includes two main modules: a web interface and a backend. These modules are very tightly coupled and they can be used separately. The former is not useful inside the PROM architecture due to the more advanced interface required by the whole system; the latter is a command line tool that requires a set of source code files and produces a file with the data of the analysis. Due to this architecture, the integration of WebMetrics tool in the PROM architecture is through the exchange of files.

5 THE IMPLEMENTATION

The PROM system includes the PROM core and tool-specific plug-ins (Fig. 1). The former is entirely written in Java, the latter are developed using different languages because Java is not always supported to write tools extensions (e.g. in Microsoft’s products).

The PROM core is completely based on open source technologies (e.g. PostgreSQL, Apache Tomcat, etc.) and standard protocols (e.g. XML [9], [21], SOAP [18], etc.). It includes:

1. **PROM Database**: it is implemented using the open source DBMS PostgreSQL.
2. **PROM Server**: it provides access to the database and exposes its functionalities as web services. It is based on Java Servlets [6] and implemented with the Apache Tomcat extended with the Apache Axis SOAP server.
3. **Plug-ins Server**: it collects data from plug-ins and performs data pre-processing. It caches information and sends data to the PROM Server when the client is online. It is implemented in Java and uses the Apache Axis libraries to communicate with plug-ins and with the PROM Server.

The PROM Server architecture (Fig. 5) runs on the Apache Tomcat 4.1 Servlet engine integrated with the Apache Axis 1.0 to manage SOAP connections. The server includes a Core and a set of pluggable Commands. All the functionalities of the system are managed through specific Commands (i.e. authentication, data collection, etc.) while the Core provides only support to the execution of the commands. The Core is implemented as a Java Servlet that launches the execution of a specific command using the parameters passed through the SOAP connection. The commands are simple java classes implementing the Command interface that can be added to the system without any change to the Core.

The SOAP protocol implements all communications between PROM components except between the PROM Database and the PROM Server that uses the JDBC protocol. The PROM server exposes a set of services that are accessible inside an Intranet or on the Internet as web services. The available implementation of the system is designed to run inside an Intranet or on the Web using a Virtual Private Network (VPN) because no encryption is implemented in the communication yet.

PROM Server – Plug-ins Servers communications use the same scheme based on the Command design pattern (Fig. 6) [8].

The chosen communication scheme provides an easy mechanism for extending the functionalities implemented through servers. The server exposes only one method (executeCommand) that takes a Command as input and produce a Result as output. Subclasses of Command implement the desired functionality and use Parameter subclasses for parameterization purposes.

Moreover, inside PROM there is the core of the WebMetrics tool that performs metrics extraction. Language parsers, that extract software metrics and are command-line executable, compose this core. For this reason, the interaction is managed through files exchange. The Plug-ins Server create a file containing the source code; then, calls a parser to analyze it; finally, extracts results from the generated file and sends them to the PROM Server that stores them into the database.

The PROM Server offers a set of basic commands such as:

- **AuthenticateCommand**: provides support to authenticate users
- **ProcessEventCommand**: collects all process metrics
- **ProcessEventMetricsCommand**: collects all code metrics
• ProcessSQLCommand: queries the database with a user-defined query and returns a result set
• ProjectMembershipCheck: checks if a user belongs to a specific project
• ReadProjectsCommand: returns the name of all projects

The PROM architecture supports third-parties tools for a more accurate data analysis. The PROM Server can provide data to advanced analysis tools that support the SOAP protocol for data exchanges.

The tool also supports manual data insertion through a web page. This modality of data collection supports non computer-oriented activities like reading manuals or paper documentation. Manual data collection is a well known error-prone activity but if it is very focused and reduced to just a few items it should be correct enough and then useful. To support this thesis, a monitoring about how developers use this particular feature will be started as soon as the tool will be used in a real environment. However, to prevent result errors, the analysis tool can discard such data.

The manual data collection is performed through web forms (Fig. 7) that collect a set of information very close to the one collected through plug-ins: users, project name, activity, starting time, and length.

6 Conclusion

This paper presented the architecture and the Java implementation of a distributed tool for collecting and analyzing software metrics and PSP data.

The collection of such data should help both developers and managers. The formers can keep track of their performances and compare them with other people in the same working group; the latter can easily keep track of projects evolution and accounting information.

The system is still under development but plans for the next version are nearly completed. That new release will provide a secure communication protocol between PROM Server and Plug-ins Servers to allow Internet use. Moreover, improved analysis tools will be integrated into the system.

Acknowledgment

This work has been co-funded by the Italian Government through the Founds for Basic Research (MAPS project, an Italian acronym for Agile Methodologies for Software Production). We would like to thank out project partners for their stimulating input.

The first author thanks Professor Philip Johnson and his research staff at the University of Hawaii at Honolulu for showing him the Hackystat system and answering his many questions.