A non-invasive approach to product metrics collection

Marco Scotto a,*, Alberto Sillitti b, Giancarlo Succi b, Tullio Vernazza a

a DIST – University of Genova, Via Opera Pia, 13 I-16145 Genova, Italy
b Free University of Bolzano/Bozen Piazza Domenicani, 3 I-39100 Bolzano, Italy

Available online 4 August 2006

Abstract

Software metrics are useful means in helping software engineers to develop large and complex software systems. In the past years, many software metrics have been proposed in order to represent several different concepts such as complexity, coupling, inheritance, reuse, etc. However, this requires the collection of large volumes of metrics and, without flexible and transparent tools, is nearly impossible to collect data accurately. This paper presents the design and the implementation of a tool for collecting and analyzing product metrics in a non-invasive way.

Keywords: Product metrics; Object-oriented metrics; Non-invasive systems

1. Introduction

Software products are becoming more and more complex; projects involving millions of lines of code are no longer exceptional. Such systems are beyond the abilities of individuals to comprehend and their evolution poses further challenges. Software metrics, which are quantitative measures of specific attributes of software development, including software process and software product, have been suggested as useful means to help software engineers to develop large and complex software systems. Prior research has discussed the importance of non-invasive data collection procedures. Pfleeger [29] discusses in detail the importance of smooth and non-intrusive data collection procedures in determining the success of a metrics program. Daskalantonakis [7] and Offen and Jeffery [27] also make similar points. Optimized data collection procedures affect metrics programs in two ways. First, they do not significantly reduce programmers’ productivity, which typically creates resistance and reluctance to establish metrics programs. Second, they increase the accuracy and reliability of the data, thereby increasing the confidence of managers who use this information. Moreover, prior literature in the area has emphasized the need of automated tools to gather, organize, and analyze metrics [12, 27]. Such tools should have a positive influence on the success of metrics programs. Their use reduces the overhead of data collection and makes analysis more accurate. In addition, the reduction of human error in data collection increases the
perception of managers of the validity of metrics information. The need not to bother the development team with what it seen as unnecessary bureaucracy is strongly felt especially within the groups following Agile methods [26]. Among the wide set of principles, guidelines, and good practices (worth to be mentioned the “barely sufficient use of documentation”), Agile Methodologies implement in software development some of the guidelines of Lean Production, pioneered by Ohno at Toyota [28]. The approach is based on a constant improvement of the production process through the reduction of “muda”: the word is the japanese term for “waste” and indicates any activity which absorbs resources but creates no value. Examples of “muda” are production of unnecessary items, processing steps which are not actually needed, movement of resources (goods or employees) from one place to another without any purpose, people in a downstream activity standing around and waiting because an upstream activity has not delivered on time, goods and services which do not meet the needs of the customer, and so on. According to this point of view, within Agile Methodologies, invasive measures may be considered part to the “muda” of the software process because they do not provide any value to the customer. Krebs [21] says that: “It is not appropriate to track lightweight processes with heavyweight metrics”.

This paper deals with the design and the implementation of an automatic tool for collecting and analyzing product metrics in a non-invasive way. The aim is to collect data and to deduce measures from such data with the minimal possible user interaction. The article is organized as follows: Section 2 introduces product metrics and problems related to their collection and analysis; Section 3 describes related work in the field of software metrics collection; Section 4 describes WebMetrics, a tool for product metrics collection; Section 5 describes PROM (PRO Metrics), a tool designed to collect and analyze process and product metrics and describes how WebMetrics has been incorporated into PROM. Finally, Section 6 draws the conclusions.

2. Product metrics

Product metrics measure any artifact or document that results from the software development process [9]. Products are not restricted to the items that management is committed to deliver to the customer. Any artifact or document produced during software life cycle can be measured. There are two kinds of product metrics:

1. Dynamic metrics, which are collected by measurements made of a program in execution.
2. Static metrics, which are collected by measurements made of the system representations such as design diagrams, source code, or documentation.

Different type of metric are related to different quality attributes. Dynamic metrics can be useful to assess the efficiency and the reliability of a program; static metrics can be useful to understand the complexity, understandability, and maintainability of a software system. Generally, dynamic metrics are easier to measure and directly related to software quality attributes than static metrics. For instance, it is quite easy to measure the execution time required for a particular function or the time required to start up a software system. These measures are directly related to the system efficiency. The number of system failures and the type of failure can be logged and related directly to the reliability of the system [39]. On the other hand, static metrics have an indirect relationship with quality attributes. A large number of static metrics have been proposed and many experiments carried out to derive and validate the relationships between these metrics and system complexity, understandability, and maintainability.

2.1. Product metrics collection issues

Although, the field of software measurement has made many advances, several issues still have to be addressed. Examples of these issues are the following:

1. Some metrics lack of a precise definition. Even straightforward metrics, such as the number of methods in a class can be difficult to define precisely. The details are often very significant (e.g., are constructors, inherited, overloaded, overridden, private, etc. methods included?) and it might be hard to change metrics collection tools in order to accommodate new definitions.
2. Data integrity has been a long-term concern for empirical software engineering and metrics researchers. In practice, available tools are rarely able to guarantee the correctness, completeness
and self-consistency of data. Sometimes, potentially important data is ignored. For instance, how Java’s anonymous inner classes contribute to the cyclomatic complexity [25] of the methods which contain them.

3. Information overload causes a loss of productivity that results from the huge amount of data generated by collecting many metrics on large projects. Information and visualization techniques are useful means to present multiple variables and assist the user by giving an intuitive view of the system.

3. Related work

There are several tools to address the problem of metrics collection. Some of them are commercial products such as CodePro Studio [6] or RSM [30]. Others are research tools such as Hackystat [11] developed at University of Hawaii.

Hackystat is able to collect both process and product metrics, however, in this section, we describe it only from the point of view of product metrics collection. Hackystat is an experimental tool for personal metrics validation. Hackystat makes available to developers a set of custom sensors that they attach to such development tools as their editor, source code control system, unit-testing framework, and so on. Once installed, these sensors automatically monitor characteristics of the developer’s process and products and send data using SOAP protocol [33] to a centralized web service. The web service maintains a repository of process and product data for each developer, performs analyses on the repository, and automatically sends the developer an e-mail when new, unexpected, and potentially interesting analysis results become available. Hackystat does not use a relational database as back-end. Instead, data for each user for each sensor data type is stored in a “log” file in XML format. In order to collect product metrics, Hackystat uses two sensors: LOCC and CCCC [4]. LOCC sensor is an Ant [2] task that sends structural metrics produced by LOCC to the Hackystat server. LOCC is a tool for generating structural metrics through source code analysis. LOCC provides support for Java and C/C++. The actual structural metrics sent depends upon the language being analyzed. They include several size metrics such as lines of code (commented and non-commented), number of methods, size in bytes of the source code, number of statements, etc. CCCC sensor is an Ant [2] task that invokes CCCC on a set of C++ files, then it send the results to the Hackystat server. CCCC is able to calculate C++ structural metrics such as Chidamber and Kemerer’s metrics suite [5], McCabe’s Cyclomatic Complexity [25], and metrics related to information flow [15].

Another interesting open-source tool is the Eclipse Metrics plug-in [8], it collects metrics and performs statistical analysis on them at development time. Metrics collected include lines of code (LOC), cyclomatic complexity [25], object-oriented metrics [5, 14], and coupling metrics [24]. Metrics can be calculated at different levels of granularity: project, source folder, package, or compilation unit. The results can be exported to an XML file for historical purposes. The tool is also able to provide a visualisation of the dependencies between packages and classes. The visualization differentiates between regular and cyclic dependencies. It highlights cyclic dependencies by allowing the user to analyze and search for all classes involved in them. The main advantage of this tool is the tight integration with the Eclipse development environment, but this, at the same time, represents the major drawback because the tool supports only the Java language.

4. WebMetrics

WebMetrics [31] is an automatic tool for product metrics collection. Metrics can be extracted from different source languages: C/C++, C#, Java, and more to come. The system is completely written in Java and parsers have been developed using JavaCC [18]. The metrics collection process requires that all code must be syntactically correct and all include files or libraries used must be present. Therefore, it must be possible to compile it without any error. At present, the tool is able to extract the following kinds of product metrics:

- Procedural metrics.
- Object-oriented metrics.

Procedural metrics measure internal attributes of functions and procedures, therefore they are applicable to almost all programming languages. In particular, the following metrics are considered: Lines of Code, McCabe Cyclomatic Complexity [25], Halstead volume [13], Fan-in and Fan-out [15].

1 http://www.eclipse.org
Many object-oriented metrics have been proposed [22,23]. One of the most referenced metrics suite is proposed by Chidamber and Kemerer (CK metrics) [5]. CK metrics suite comprises six metrics, which capture different aspects of object oriented systems, including complexity, coupling, and cohesion. The suggested metrics are: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object classes (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM).

4.1. JavaCC

JavaCC [18] is a parser generator, which takes in input an attributed grammar and produces a scanner and a recursive descendent parser [38]. The user has to add classes (e.g., for symbol table handling, optimization and code generation), whose methods are called from the semantic actions of the attributed grammar.

Attributed grammars [20] were introduced as a notation for describing the translation of languages. Originally they were declarative in nature, however, JavaCC uses them as procedural descriptions. In this form, an attributed grammar consists of the following parts:

- a context-free grammar using Wirth’s EBNF (Extended Backus Naur Form) [37]. It describes the syntax of the language to be processed.
- attributes that can be considered as parameters of the non-terminal symbols in the grammar. Input attributes provide information from the context of the non-terminal, while output attributes deliver results computed during the processing of the non-terminal.
- semantic actions, which are statements in an imperative programming language (e.g., Java) that are executed during parsing. They compute attribute values and call methods e.g., for symbol table handling or code generation. In JavaCC, semantic actions are enclosed by “{” and “}”.

Every production of an attributed grammar is processed from left to right. While the syntax is parsed semantic actions are executed where they appear in the production.

Parsers generated by JavaCC are based on a simple, efficient, and convenient parsing technique for integrating semantic processing. However, it requires the grammar of the parsed language to be LL(1), which means that the parser must always be able to select between alternatives with a single symbol lookahead. Unfortunately, many important programming languages such as Java, C++, or C# are not LL(1), so that one either has to use a bottom up LALR(1) [1,19] parsing technique, or the parser has to resolve LL(1) conflicts using semantic information or a multisymbol lookahead. JavaCC offers various strategies for resolving LL(1) conflicts: first, one can instruct JavaCC to produce a LL(k) parser with $k > 1$, i.e., maintaining $k$ lookahead symbols all time. Many language, however, not only fail to be LL(1), but they are not even LL(k) for any arbitrary, but fixed $k$. Therefore, JavaCC offers local conflict resolvers (keyword: LOOKAHEAD(…)) that can be placed in front of alternatives. A conflict resolver is a boolean expression that is inserted into the grammar at the beginning of the first of two conflicting alternatives and decides by multisymbol lookahead or by semantic checks whether this alternative matches the actual input. If the resolver yields true, the alternative is selected, otherwise the next alternative will be checked. Moreover, JavaCC allows an increase in the number of lookahead symbols locally (e.g., LOOKAHEAD(2)) or to provide an arbitrary EBNF expression that will be compared with the input.

4.2. Description of the architecture

WebMetrics uses a set of relations as an intermediate representation in order to decouple software metrics extraction from their use. Parsers simply generate a set of intuitive relations, which a separate analyzer uses as input to compute arbitrary metrics. Relations describe the interaction among different language entities, such as classes, interfaces, methods, and attributes. This relations set is stored in a database. Therefore, metrics can be calculated directly by querying the relation set using the SQL language. The architecture includes three main components (Fig. 1):

1. WebMetrics Parsers: Each parser consists of a grammar parser, a symbol table, and a set of supporting classes. The grammar parser recognizes the syntax of a particular language and it is written by using JavaCC [18]. The symbol table and all supporting classes are written in Java language. Parsers share a module called Common Interface, which provides a standard API to report relations on the database.
2. **Database**: It stores the relation set that represents the source code. The implementation is based on the open source DBMS MySQL.

3. **Analyzer**: It calculates the metrics by querying the relation set with the SQL language. Each metric is a class that implements the `Measure` interface. This is implemented according to the `Strategy` design pattern [10] therefore; the analyzer can calculate each metric in exactly the same way through polymorphism. New metrics can be developed by simply constructing a new class implementing this interface.

At run-time, the typical execution pattern is the following: the `Parser` parses the source code and populates a symbol table. Then, each entry in the symbol table is asked to report its relations on the database; the report system is implemented according to the `Visitor` design pattern [10]; finally, the `Analyzer` queries the relation set stored in the database to calculate the desired metrics.

5. **PROM (PRO Metrics)**

PROM [32] is a distributed architecture designed to collect different set of software data: software metrics and Personal Software Process data [16]. PROM provides plug-ins for mass-market applications such as office automation, popular IDEs, etc. In this way, process metrics collection is completely transparent and users do not have to learn new applications (Fig. 2). PROM is component-based; more specifically, it is based on the Package-Oriented Programming development technique [34]. The architecture design fullfills three main requirements:

1. The architecture should be extensible to support new IDEs, new programming languages, kinds of data, and analysis tools.
2. IDE dependent plug-ins need to be as simple as possible.
3. Developers can work off-line.

The PROM core is completely written in Java and uses open source technologies and standard protocols such as XML and SOAP (a lightweight protocol based on XML, in our case over HTTPS channels [40]).

5.1. **Process metrics collection**

The architecture devoted to the collection of process metrics includes four main components (Fig. 2):

1. **PROM plug-ins** are IDE dependant plug-ins that listen to application-generated events such as file open, file save, edit, etc. They collect and send all these data to the `PROM Transfer`, adding timestamp information and providing user authentication features to identify users that are working. This authentication feature is different from the traditional ones in that multi-user logins are allowed, so as to support “Agile” software development practices, such as pair programming (two developers working in front of the same machine).
2. **PROM Trace** is a specific plug-in that allows to track the most interesting calls of the operating system and, therefore, the interaction of the user with the system: the level of logging is customizable and allows to track in detail the user activity, such as name of the running application, title of the window which has the focus, open/close file activities and so on.
3. The `PROM Transfer` collects the data from all the plug-ins and makes some redundancy reduction pre-processing. `PROM Transfer` sends these results to the PROM Server that stores them into the database. This component provides also off-
line working capabilities: it collects all data in the cache and sends them to the PROM Server when it is connected.

4. The PROM Server provides an interface to the PROM Database through high-level commands, exposed as SOAP services. This interface hides completely the DBMS and the low-level data model. This server provides all its functionalities through Web Services. The implementation is based on the Apache Tomcat application server, integrated with the Apache Axis SOAP server [3] and uses the JDBC technology for the database access.

5. The PROM Database contains all acquired data and information on users and projects. The implementation is based on the open source DBMS PostgreSQL.

5.2. Product metrics collection

WebMetrics has been incorporated into a PROM component called Product Manager (PM). PROM PM is able to track code evolution by keeping track of versions of source code into a CVS repository. When a piece of code is stored into a version control system, PM invokes the software metrics extractor (WebMetrics) that collects metrics data and stores them into the PROM Database. This feature provides a comprehensive view of the source code evolution during the development process. The main components of this architecture (Fig. 3) are:

1. PROM Database: It stores all acquired product metrics. The implementation is based on the open source DBMS PostgreSQL.

2. PROM Server: It provides an interface to the PROM Database through high-level commands exposed as SOAP services [33]. This interface hides completely the DBMS and the low-level data model. The implementation is based on the open-source SOAP engine Axis [3].

3. PROM Product Manager (PROM PM): This component orchestrates the whole process of product metrics acquisition. In particular, it checkouts the last version of source code from the CVS repository, it invokes WebMetrics in order to calculate product metrics, and, finally, it uploads metrics to the PROM Database using the SOAP interface exposed by the PROM Server. The communication between PROM PM and PROM Server uses the HTTPS protocol in order to guarantee confidentiality on product metrics collected.

5.3. PROM Database

Data collection model is a key part of the PROM system because it contains data regarding users, projects, collected events, and product metrics. Relations between elements are also stored here.
Fig. 4 shows a partial view of the PROM data model, in particular it only depicts the tables involved in the collection of product metrics. Mainly, PROM PM stores product metrics into the $wm_{\_}webmetrics$ table. This table stores information about the filename where the metric is collected, the kind of the entity on which the metric is calculated on (it can be a class, a method, or an interface) and the value of the metric itself. Other supporting tables make it possible to attach to every measure the type of metric ($wm_{\_}metrics$ table), a timestamp ($uploads$ table), a project ($projects$ table), and the programming language of the source code analyzed ($wm_{\_}languages$ table).

### 5.4. PROM Server

PROM Server and the Product Manager communicate using the SOAP protocol. The communications schema is based on the Command design pattern [10]. The PROM Server exposes only one function (executeCommand) that accepts a Command object and returns a Result object. The command can contain a Parameter, both of them are generated at the same time through the CommandBuilder class that implements the Builder design pattern [10]. PROM Server provides several commands, but PROM PM invokes only a subset of these:

- **GenerateUploadIdCommand**, returns a timestamp.
- **MetricsCommand**, uploads product metrics.
- **GetUserIdCommand**, given the user name, it returns the corresponding user id.
- **GetProjectIdForCommand**, returns the name and the corresponding id of all projects.

### 5.5. PROM Product Manager

PROM Product Manager (PROM PM) is a standalone Java application that can be scheduled to collect product metrics on a regular basis. In particular, it manages the product metrics acquisition process and it is in charge of uploading such metrics to the PROM Server. It performs the following steps:

1. Checkout of the last version of source code; PROM PM supports the CVS version control system. It is also able to connect to CVS repositories that use the SSH (Secure Socket Host) protocol to encrypt data.
2. Compile source code; this step is only necessary to parse Java or C# code because these parsers use the reflection API of the corresponding language for resolving references to classes and methods. This feature is particularly useful for resolving symbols for which source code is not present, such as standard library classes and third party libraries.
3. Invoke WebMetrics in order to calculate product metrics.
4. Upload calculated product metrics to the PROM Server. This functionality exploits the Web Service API provided by Axis [3]. We choose the Web Services standard because it relies on HTTP protocol and, therefore, it can work through many common firewall security measures without requiring changes to their filtering rules.

### 6. Conclusions

This paper described an architecture for collecting and analyzing product metrics in a non-invasive
way. The current version of the system does only support CVS as source code repository, but we plan to support other version control systems such as: Microsoft Visual Source Safe [36], IBM Clear Case [17], and Subversion [35]. We also plan to extract information from version control systems because they indirectly collect data regarding the software development process.

References

[1] A.V. Aho, J.D. Ullmanm, Principles of compiler design, Addison-Wesley, 1977.
[2] Apache Ant. Available from: <http://ant.apache.org/>.
[3] Apache Axis. Available from: <http://ws.apache.org/axis/>.
[4] CCCC (C and C++ Code Counter). Available from: <http://cccc.sourceforge.net/>.
[5] R. Chidamber, F. Kemerer, A Metrics Suite for Object-Oriented Design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[6] CodePro Studio. Available from: <http://www.instantiations.com/codepro>.
[7] M. Daskalantonakis, A practical view of software measurement and implementation experiences within motorola, IEEE Transactions on Software Engineering 18 (November) (1992) 998–1010.
[8] Eclipse Metrics Plugin. Available from: <http://metrics.sourceforge.net/>.
[9] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, Thomson Computer Press, 1994.
[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[11] Hackystat, University of Hawaii. Available from: <http://csl.ics.hawaii.edu/Research/Hackystat/>.
[12] T. Hall, N. Fenton, Implementing effective software metrics programs, IEEE Software (March/April) (1997) 55–65.
[13] M.H. Halstead, Elements of Software Science, Elsevier, Amsterdam, 1977.
[14] B. Henderson-Sellers, Object-Oriented Metrics, measures of Complexity, Prentice-Hall, 1996.
[15] S.M. Henry, D.G. Kafura, Software structure metric based on information flow, IEEE Transactions on Software Engineering 7 (5) (1981) 522–545.
[16] W. Humphrey, A Discipline for Software Engineering, Addison-Wesley, 1995.
[17] IBM Clear Case. Available from: <http://www-306.ibm.com/software/awdtools/clearcase/>.
[18] JavaCC (Java Compiler Compiler). Available from: <http://javacc.dev.java.net/>.
[19] D.E. Knuth, On the translation of languages from left to right, Information and Control 6 (1965) 607–639.
[20] D.E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (1968) 127–145.
[21] W. Krebs, Turning the knobs: a coaching pattern for XP through Agile metrics, XP/Agile Universe (2002) 60–69.
[22] W. Li, Another metric suite for object-oriented programming, Journal of Systems and Software 44 (1998) 155–162.
[23] M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice-Hall Inc., 1994.
[24] R.C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice-Hall, 2002.
[25] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering 30 (4) (1976) 308–320.
[26] M. Marchesi, G. Succi, Extreme programming and Agile processes in software engineering, in: Proceedings of XP, 2003.
[27] R.J. Offen, R. Jeffery, Establishing software measurement programs, IEEE Software (March/April) (1997) 45–53.
[28] T. Ohno, Toyota Production System: Beyond Large-Scale Production, Productivity Press, Portland (OR), 1988.
[29] S.L. Pfleeger, Lessons learned in building a corporate metrics program, IEEE Software (May) (1993) 67–74.
[30] RSM (Resource Standard Metrics). <http://msquaredtechnologies.com/>.
[31] M. Scotto, A. Sillitti, G. Succi, T. Vernazza, A relational approach to software metrics, in: Proceedings of the 19th Annual ACM Symposium on Applied Computing, Nicosia, Cyprus, 2004, pp. 1536–1540.
[32] A. Sillitti, A. Janes, G. Succi, T. Vernazza, Measure for mobile users: an architecture, Journal of System Architectures 50 (7) (2004) 365–444.
[33] Simple object access protocol (SOAP) specification. <www.w3.org/TR/soap/>.
[34] G. Succi, W. Pedrycz, E. Liu, J. Yip, Package-oriented software engineering: a generic architecture, IEEE IT Pro (April) (2001).
[35] Subversion. <http://subversion.tigris.org/>.
[36] Microsoft visual source safe. <http://msdn.microsoft.com/ssafe/>.
[37] N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions, Communications of the ACM (1977).
[38] N. Wirth, Compiler Construction, Addison-Wesley, 1996.
[39] A. Wood, Predicting software reliability, IEEE Software (November) (1996) 69–77.
[40] XML Protocol working group. Available from: <http://www.w3c.org/2000/xp/Group/>.