Empirical Project Monitor: Automatic Data Collection and Analysis toward Software Process Improvement

Masao Ohira Reishi Yokomori Makoto Sakai

Ken-ichi Matsumoto Katsuro Inoue Koji Torii

In recent years, improvement of software process is increasingly gaining attention. However, its practice is very difficult because coherent data collection and utilization of the collected data require considerable experience with software process improvement. In this paper, we describe our empirical approach to software engineering and introduce Empirical Project Monitor (EPM). Collecting data on development activities from common software development support tools such as configuration management systems and mailing list managers, EPM analyzes the stored data automatically and provides graphical results. EPM facilitates coherent data collection and data analysis which are difficult tasks in practice.

1 Introduction

In software development in recent years, improvement of software process is increasingly gaining attention. Its practice in software organizations consists of repeatedly measuring the development activities, finding potential problems in the processes, assessing improvement plans, and providing feedback into the processes [1].

Many software measurement methods have been proposed to better understand, monitor, control, and predict processes and products. However, they cannot be used effectively to improve processes unless people have extensive experience with software measurement because it is difficult

- to specify an explicit goal for improvement,
- to prepare necessary software or metrics for measurement,
- to bring measurement results into further process improvement, and so on.

Moreover, since consecutive practices of collecting data often canceled by replacement of a manager or developer, sufficient experience with measurement is not easy to be accumulated in organizations.

As an approach to deal with the issues above, the Goal-Question-Metric (GQM) paradigm [2] [3] [4] [5] provides a sophisticated measurement technique. GQM guides to set up measurement goals, create questions based on these goals, and determine measurement models and procedures based on these

Masao Ohira, Ken-ichi Matsumoto, Koji Torii, Nara Institute of Science and Technology

Reishi Yokomori, Katsuro Inoue, Osaka University

Makoto Sakai, SRA Key Technology Laboratory, Inc.

questions. The improvement of software process based on GQM is a logical and reasonable method. However, in its practice, it is necessary for all members to strive about all measurement processes on every last detail, so honest efforts and high costs are needed.

In summary, as practical issues, measurement efforts do not succeed in improving software process effectively, because

(1) little measurement experience and

(2) the burden of strict measurement

result in giving up continuous, coherent measurement activities in many cases.

We have been studying empirical software engineering [6] [7] [8] as a measurement-based approach to these problems. Empirical software engineering is a method of evaluating various technologies and tools based on the quantitative data obtained through actual use. In our study, we are trying to develop a variety of support tools called Empirical software Engineering Environment (ESEE), which helps developers/managers/organizations improve software processes. Recently we have developed Empirical Project Monitor (EPM) as a partial implementation of ESEE.

EPM automatically collects development data from common development support systems, which are often used in recent open source software development, such as CVS[†1] and Mailman[†2]. EPM also analyzes the collected data and provides users with graphical results. Using EPM, users can obtain objective data at low cost and keep current development status under control. Moreover, since EPM helps developers/managers share the results through using a web browser, it is easy to construct a shared environment for discussing on the collected data and the analyzed results.

†1 CVS, Concurrent Version System, http://www.cvshome.org/

†2 Mailman, the GNU Mailing List Manager, http://www.list.org/

Section 2 describes our approach to tackle issues on software process improvement and Empirical Software Engineering Environment (ESEE). We introduce Empirical Project Monitor (EPM), which has been developed as a partial implementation of ESEE, in Section 3. Section 4 illustrates some of the advantages of using EPM. Section 5 summarizes this paper and future work.

2 Empirical Software Engineering Environment (ESEE)

We describe here that our approach to dealing with the issues mentioned the previous section. First, Section 2.1 introduces the general cyclic model for software process improvement, which is fundamental concept in our empirical approach to software engineering. Section 2.2 describes our policies to constract a computational environment that supports to practice the continuous, cyclic process in software development organizations. Section 2.3 is the architecture of Empirical Software Engineering Environment (ESEE) which provides a combination of computational support tools for coherent deta collection and analysis.

2.1 The Cyclic Process for Improvement

Many models related to software process improvement have been proposed (e.g. CMM [9], CMMI [10], IDEAL MODEL [11], etc.). These models have common characteristics which provide a series of guidelines in order to raise productivity and reliability of software through continual, gradual improvement. It is essential to understand (assess) current status and problems properly as a starting point for applying the models in practice.

Instead of trying to develop "another CMM", our approach is to provide a computational environment for supporting quantitative data-based measurement, which can complement to practice ex-

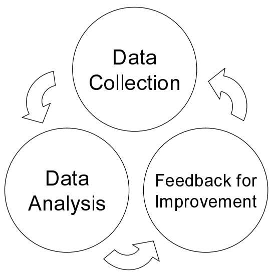


Figure 1 Basic model for improvement

isting models such as CMM. Figure 1 is roughly sketched our model for process improvement as follows:

1. massive data collection from software development activities
2. Intensive data analysis
3. Feedback for software process improvement

Our model doses not suppose rigid procedures such as other models but flexible improvement processes according to respective problems in software development projects and organizations. To provide a computational environment to support such flexible improvement, we think that it is important to be able to (1) measure (assess) software process objectively based on coherent quantitative data without heavily depending on individual experience, and (2) keep on conducting measurement activities without additional work and costs. These are also challenges to the two issues mentioned in Section 1.

2.2 Policies for Designing ESEE

Though we do not specify procedures for improvement activities, we are trying to design a computational support environment for quantitative data collection and analysis, according to the three policies as follows.

Policy for Data Collection: As we mentioned in Section 2.1, our approach does not intend to collect data after specifying a clear-cut goal for improvement or analysis. Our approach gives priority to collect as much data as possible coherently without disturbing ordinary development activities. Similar approaches [12] [13] have proposed to analyze histories of software changes by using rapidly growing hardware capabilities.

- Not goal first (ideal cases) but data collection first (realistic approach) because it is difficult to determine a goal for improvement at first in practice
- Collect mainly product data (process data is obtained from the product data) in order to minimize developers/managers' overhead for collection
- Collect raw data without human tampering in order to prevent involving subjective data such as human-handed documents
- Real-time data collection in order to keep development activities under control
- Make applicable to various projects (e.g. small scale projects, non-water fall process projects such as XP, distributed development projects including sub-contracting, etc.) by collecting data from widely used development support systems such as configuration management systems and mailing list managers.

Policy for Data Analysis: Our main target in supporting process improvement is thousands of projects and organizations that must manage many projects, rather than a single project or individual developers/managers. Although there are many studies on development support in a small or middle scale, few have been proposed to support software development projects and organizations in a large scale. Since there is few technology and technique for analyzing such large-scale software development data, we are doing stepwise implementation such as Table 1.

Section 3 introduces Empirical Project Monitor (EPM) which supports 1 in Table 1 (, and 2 in the

Table 1 Analysis policy for stepwise implementation

Priority	Technologies to analyze large-scale development data
1	Process/product metrics inside a single project (easy, simple)
2	Inter-project metrics
3	Classification and evolution
4	Reuse components/expertise
5	Deeper analysis (difficult, complex)

near future).

Policy for Improvement: We would like to provide flexible feedback methods for each objective in projects and organizations. Currently we are evaluating existing technologies and inventing a mechanism in order to deal with various cases. We have been designing Empirical Software Engineering Environment (ESEE) while incorporating such a flexible mechanism in ESEE. Section 2.3 describes the architecture of ESEE.

2.3 The Architecture of ESEE

Empirical Software Engineering Environment (ESEE) is not a single huge system that supports all the steps as in Figure 1 but a flexible system with various pluggable tools, which can be replaced according to objectives of improvement and methods of data analysis in organizations.

The fundamental functions of ESEE are:

- to collect a large amount of data from thousands of software projects and
- to make such the huge data available to analyze for improving processes and increasing organizational benefit.

Figure 2 shows the system architecture of ESEE. ESEE mainly consists of the four parts (data collection, format translation, data store, and data analysis/visualization). In what follows, we describe these parts briefly.

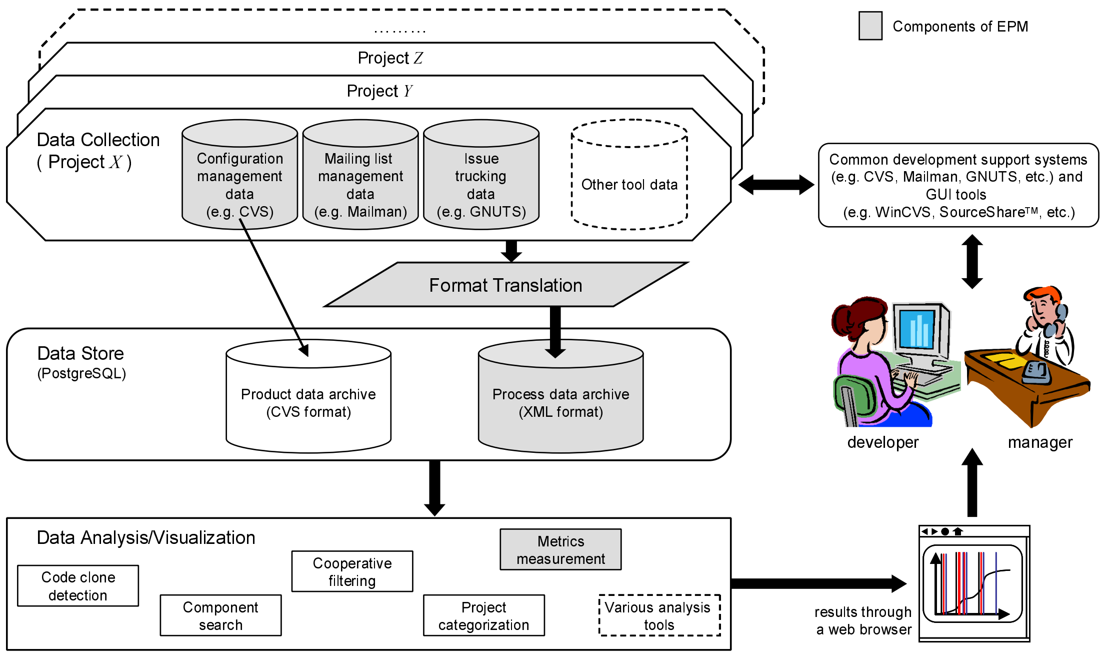
Data collection: In the data collection part, data from thousands of software projects is automatically gathered through common software development support tools such as configuration management systems and mailing list managers. For instance, the data from a configuration management system contains information relevant to software development activities (e.g. "who/when/how changed the source code," "how many times the source code was changed," and so on). Data collection from widely used development support systems prevents developers and managers from increasing additional work for measurement.

Format translation: In the format translation part, the collected data are classified into product data in the CVS format and into process data in the XML format which we call the standardized empirical software engineering data format. According to the data collection policy, ESEE collects three kinds of data from configuration management systems, mailing list managers, and bug reports management systems. Various kinds of data form other systems are also available according to the purposes of data analysis if the data are transformed into the XML format.

Data store: The data is stored in two PostgreSQL†3 databases. One is for the product data which is necessary to analyze source codes such as code clone detection [14]. Another is for the process data to analyze development activities.

Data analysis/visualization: We have plans to provide a variety of analysis/visualization tools,

†3 PostgreSQL, the worlds most advanced Open Source database software, http://www.postgresql.org/



Components of EPM
………
Project Z
Project Y
Data Collection (Project X)
Configuration management data (e.g. CVS)
Mailing list management data (e.g. Mailman)
Issue trucking data (e.g. GNUTS)
Other tool data
Common development support systems (e.g. CVS, Mailman, GNUTS, etc.) and GUI tools (e.g. WinCVS, SourceShare™, etc.)
Format Translation
Data Store (PostgreSQL)
Product data archive (CVS format)
Process data archive (XML format)
developer
manager
Data Analysis/Visualization
Code clone detection
Component search
Cooperative filtering
Project categorization
Metrics measurement
Various analysis tools
results through a web browser

Figure 2 Empirical Software Engineering Environment (ESEE) and Empirical Project Monitor (EPM)

which help managers and developers detect code clone, search software components [15], find similar projects by cooperative filtering [16], categorize projects [17] and so on. EPM introduced in Section 3 measures basic metrics using the data stored in three development support systems and provides users with graphical results.

In this way, ESEE provides a mechanism in order to collects massive data from thousands of projects automatically and to provide developers and managers with the analysis results. The pluggable mechanism makes ESEE flexible to extend further functions according to purposes for improvement in organizations. This feature of ESEE would be important to support various projects and organizations because purposes of data collection/analysis for improvement differ among projects and organizations. We would like to provide such the framework rather than provide a "tailored" integral system at first.

3 Empirical Project Monitor (EPM)

We have developed Empirical Project Monitor (EPM) as a partial implementation of ESEE. EPM is an automatic data collection and analysis system in order to support software process improvement. Automatically collecting data on development activities from common development support systems, EPM analyzes the stored data and provides users with graphical results. EPM assists coherent quantitative data collection and facilitate data analysis, which both are difficult tasks in practice.

3.1 Overview of EPM

EPM consists of basic components in ESEE (the grayed components in Figure 2). EPM collects the data from widely used development support systems as follows:

- configuration management systems

- mailing list managers
- bug (issue) tracking systems

Currently EPM can deal with the data from CVS (configuration management system), Mailman/Majordomo[†4]/fml[†5] (mailing list managers), and GNATS[†6]/Bugzilla[†7] (bug tracking systems). The data from other systems are also available by small adjustments if the parameters are converted into the XML format.

These data are collected automatically through using common GUIs (e.g. SourceShare[†8], WinCVS[†9], etc.), which are also used for supporting software development projects in recent years. Developers and managers do not need additional work for data collection.

EPM analyzes the process data transformed in the XML format in the PostgreSQL database, provides users (developers and managers) with various visualizations of the data. Because EPM can collect and analyze the data in real-time, EPM help users understand current states of their projects and keep their projects under control.

For example, from a CVS repository, the format translator extracts process data about events such as modification, checkout, check-in, and so on and translates the data in the XML format. At the same time, the translator produces the XML data about transition of sizes, update times and versions for each stored component in the repository. The translator also extracts reports such as bug-detection reports and bug-fix reports in a bug tracking system. Moreover, the format translator acquires information such as posted time of each mail, subjects, senders, and so on from header information stored in a mailing list archive.

The results of the data analysis are available to see through using a common web browser, so that EPM assists users in sharing the results. The easiness of sharing the analysis results would help users discuss on the results and plan further improvement points.

In this way, EPM supports users to obtain objective data at low cost and to keep current development status under control.

3.2 Visualization Results of EPM

EPM measures the collected data using metrics as follows:

- LOC (lines of code)
- Number of check-ins/checkouts
- Number of mails
- Number of bugs (issue reports)
- Development activity, and so on.

Analyzing the collected data using such metrics, EPM provides five kinds of visualizations currently.

Growth of LOC: For instance, Figure 3 shows the change of cumulative total of source codes in our own project (EASE project [18]). The vertical lines represent when developers "checked-in" the CVS repository. It means that developers changed (added/modified/deleted) a source code in the repository. The graph helps users understand the current progress of the project and estimate the future status by comparing to past similar projects.

Use of CVS by developers: Figure 4 represents the relationship between time of check-ins (vertical longer grayed lines) and number of checkouts. When a check-in occurs, other developers usually check out the CVS repository in order to update their local files. Fewer checkouts often mean

†4 Majordomo, mailing list management packages, http://www.greatcircle.com/majordomo/

†5 fml, Mailing List Server Package, http://www.fml.org/index.html.en

†6 GNATS, GNU Bug Tracking System, http://www.gnu.org/software/gnats/

†7 Bugzilla, Bug Tracking System, http://www.bugzilla.org/

†8 SourceShare, Novel Electronic Software Publishing environment, http://www.zeesource.net/

†9 WinCVS, Windows GUI front-end for CVS, http://wincvs.org/

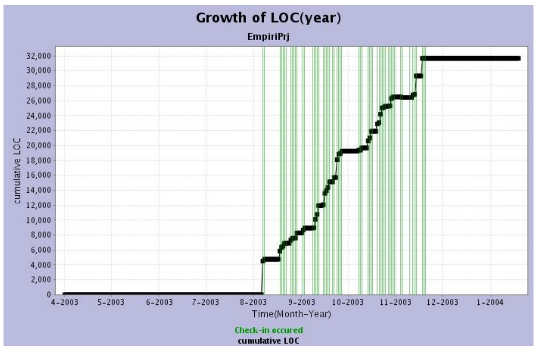


Figure 3　Growth of LOC

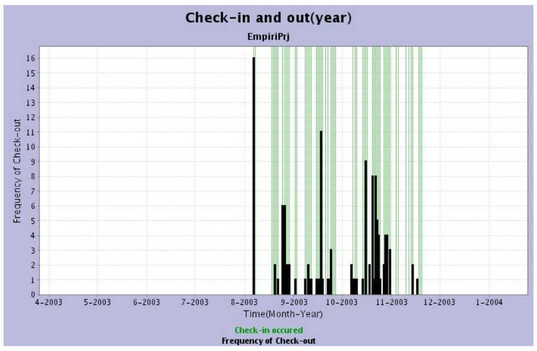


Figure 4　History of the CVS repository

that the files updated into the CVS repository are not important for other developers or others may not know the check-in occurred. The graph tells users whether developers refer to the CVS repository after check-ins or whether there exists a check-in which should be informed to others.

Communication among developers: Figure 5 illustrates the change of cumulative total of e-mails (the line graph), time of issues raised/resolved (the vertical shorter/longer dashed lines), and time of check-ins occurred (the vertical light-gray line). From this graph, users can know an overview of communications among them, since communications usually become active to discuss on the bugs when bug issues are reported to a bug tracking system. If many bugs are reported but developers do not discuss on the bugs, communications among developers may have problems. Communication problems among developers almost always bring the decrease of software productivity and reliability [19] [20].

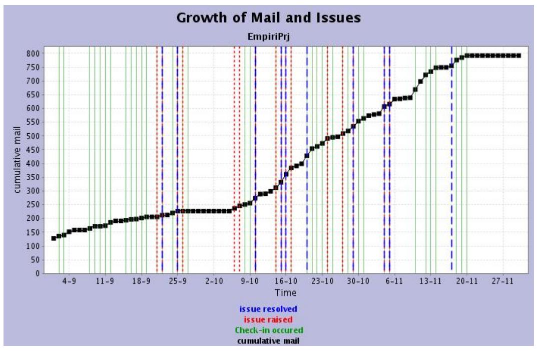


Figure 5　History of communications among developers

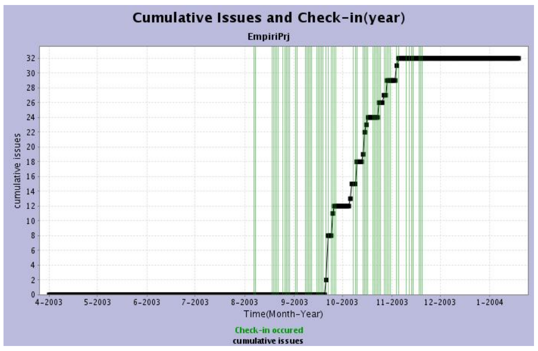


Figure 6　Relationship between issue reports and check-in

Bug issues and developers' activity: Figure 6 shows the relationship between the change of cumulative total of issues (the line graph) and time of check-ins (the grayed vertical line). Check-ins often occurs when issues are raised because developers try to modify the bugs. The graph helps users confirm the situation of issues per every version.

Unsolved bug issues in a project: Figure 7 represents the relationship among the change of cumulative total of issues (the upper line), the change of number of unsolved issues (the lower line), and change of mean time until issues are resolved (the middle line). This graph is useful for managers to

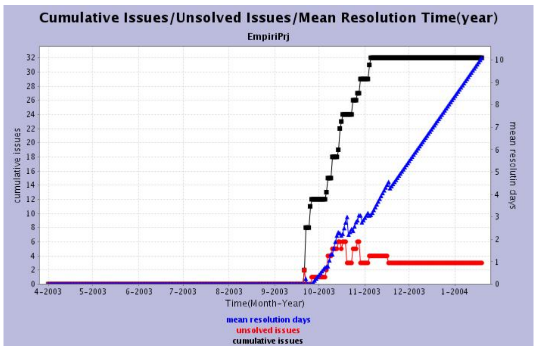


Figure 7 Unsolved issues in a project

know how long time it takes to resolve one issue at present. This also might help developers give motivations for trying to decrease bugs.

In this way, EPM visualizes useful information for understanding the current status of project and keeping projects under control. The current implementation of EPM only provides above five kinds of graphs in different scales (day/month/year). In the near future, we will implement the function to visualize multi-projects data based on the analysis policy 2 in Table 1. We will also provide users with the customizability for directly operating the PostgreSQL in order to support user's purposes for data analysis.

4 Scenario of EPM

We illustrate some of the advantages of EPM using scenarios. The followings are often mentioned as problems in practical software development.

- increase of developers/managers' burden when collecting data for process improvement
- delay of information exchanges related to projects' progress
- arbitrary human-hand operations to data

First one is an obvious problem because developers/managers are often required to do additional work such as progress reports and product reports, once starting to collect the data to measure the productivity, the reliability, and so on. It also means that the measurement takes high costs. Even if using automatic monitoring tools for measuring developers・activities, there still exist costs (e.g. All developers have to install the tools into all machines which are used for development and so on).

EPM automatically collects/analyzes the raw product data generated from existing popular tools in software development (the process data is calculated from the product data). In many cases, it is sufficient for EPM to be installed into a normal computer per an organization, without other additional tools and work. If one software project does not use CVS, it takes less cost to introduce CVS into the project because CVS is free, open source software. That is the reason why we decide to exploit the data from open source software such as CVS and Mailman, considering the easiness for the introduction of software tools.

Second one is a problem related to communications. Less communication among developers almost always causes low productivity [20]. In particular, this is critical for multi-site distributed software development in recent years because developers have little chance for face-to-face communications. We have been already using e-mails to exchange information with others. However, developers in geographically distributed environments are hard to acquire right information at right time from appropriate persons [19].

One of the characteristics of EPM is to collect data in real-time. This is useful to understand the development status at present. Especially as we introduced using Figure 5, the graph which represents history of communication are available as a kind of "alerts". Users (developers/managers) can confirm whether developers communicated sufficiently when issues occurred. If developers do not communicate (discuss) with each other in spite of issues rising, it might indicates the shortage of communications in the project.

Final is one of problems of the document (product) based project management. The quality of documents by human-hand generally has no consistency. It often depends on individuals and their arbitrary interpretations, so that a new person from different culture may change the whole quality of documents in a project differently. This often happens when one developer/manager retires from the project. Therefore, this makes reuse of their previous experience difficult.

Using EPM, such the arbitrary operations cannot be possible basically because EPM deals with the raw process data extracted from the product data, which is generated in ordinary development activities. This feature would lead continuous, coherent process improvement activities.

5 Summary and Future Work

In this paper, we introduced our empirical approach to software engineering for process improvement. Empirical Software Engineering Environment (ESEE) gives a computational framework for quantitative measurement.

Our goal of this study is to construct the methodology for supporting measurement based software development. Nowadays, we can gather and analyze massive data on software development in a large scale using rapidly growing hardware capabilities. By analyzing the huge data collected from thousands of software development projects, we would like to provide useful knowledge and benefit not only to individual developers/managers but also to organizations.

Although we much need to understand "what kinds of technologies/feedback are effective to increase organizational benefit," we believe that ESEE could strongly complement existing techniques such as GQM, because the problems found from quantitative coherent data can be used as the evidence in order to determine the goal of the improvement.

We also introduce Empirical Project Monitor (EPM) as a partial implementation of ESEE, which supports developers/managers keep projects under control by providing various visualization results related to project activities. We are implementing further functions to help users customize visualizations freely. In particular, the function to visualize multi-project data according to our analysis policy, represented by 2 in Table 1, will be available soon. We also have plans to apply EPM to practical software development projects in companies.

Empirical study on software development is an active area in the field of Empirical Software Engineering (ESE). But the approaches of ESE have not been sufficiently applied to software development in software industry in spite of holding many problems. We are trying to identify the factors from an organizational point of view.

We are collaborating with some software development companies. The data about software development from the industrial world has seldom been provided with university's research. Therefore, it would be a strong trigger for going beyond inhibition of the technical progress of software engineering. By empirical approach, we expect to break down the wall of inhibition and to advance research toward problem-solving.

Acknowledgment

This work is supported by the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology.

We thank Satoru Iwamura, Eiji Ono and Taira Shinkai for supporting development of Empirical Software Engineering Environment and Empirical Project Monitor.

Reference

[1] L. Briand, C. Differding, and D. Rombach, Practical guidelines for measurement-based process improvement, Technical Report ISERN–96–05, Department of Computer Science, University of Kaiserslautern, Germany, 1996.

[2] V. Basili and D. Weiss, A methodology for collecting valid Software Engineering Data, IEEE Transactions on Software Engineering, Vol.10, No.6, pp.728–738, 1984.

[3] V. Basili and H. D. Rombach, The TAME Project: Towards Improvement–Oriented Software Environments, IEEE Transactions on Software Engineering, Vol.14 No.6, pp.758–773, 1988.

[4] H. D. Rombach, Practical Benefits of Goal–Oriented Measurement, In N. fenton and B. Littlewood, editors, Software Reliability and Metrics, pp. 217–235, Elsevier Applied Science, London, 1991.

[5] V. Basili, Applying the Goal/Question/Metric Paradigm in the Experience Factory, Presented at the 10th Annual CSR Workshop in Amsterdam, 1993.

[6] A. Aurum, R. Jeffery, C. Wohlin, and M. Handzic, Managing Software Engineering Knowledge, Springer, Germany, 2003.

[7] V. Basili, The Experimental Software Engineering Group: A Perspective, ICSE'00 award presentation, Limerick, Ireland, June 5–10, 2000.

[8] M. Lindvall, V. Basili, B. Boehm, P. Costa, and K. Dangle, F. Shull, R. Tesoriero, L. Williams, and M. Zelkowitz, Empirical Findings in Agile Methods, Proceedings of XP/Agile Universe 2002, pp.197–207, 2002.

[9] M. C. Paulk, B. Curtis, M. B. Chrissis and C. V. Weber, Capability Maturity Model Version 1.1, IEEE Software, Vol.10, No.4, pp.18–27, 1993.

[10] CMMI Product Team, Capability Maturity Model Integration (CMMI) Version 1.1 CMMI for Systems Engineering and Software Engineering (CMMI–SE/SW, v1.1), Continuous Representation, CMU/SEI–2002–TR–001, ESC–TR–2002–001, 2001.

[11] J. Gremba and C. Myers, The IDEAL Model: A Practical Guide for Improvement, appeared in the Software Engineering Institute (SEI) publication, Bridge, issue 3, 1997.

[12] D. Draheim and L. Pekacki, Process-Centric Analytical Processing of Version Control Data, International Workshop on Principles of Software Evolution (IWPSE'02), pp.131–136, Helsinki, Finland, 2003.

[13] A. Mockus and L. G. Votta, Identifying Reasons for Software Changes Using Historic Database, Proceedings of 2000 International Conference on Software Maintenance (ICSM'00) pp.120–130, San Jose, CA, 2000.

[14] T. Kamiya, S. Kusumoto, and K. Inoue, CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, Vol.28, No.7, pp. 654–670, 2002.

[15] R. Yokomori, T. Ishio, T. Yamamoto, M. Matsushita, S. Kusumoto, and K. Inoue, Java Program Analysis Projects in Osaka University: Aspect-Based Slicing System ADAS and Ranked-Component Search System SPARS-J, Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pp.828–829, Portland, Oregon, 2003.

[16] N. Ohsugi, A. Monden, and S. Morisaki, Collaborative Filtering Approach for Software Function Discovery, Proceedings of 2002 International Symposium on Empirical Software Engineering (ISESE 2002), Vol.2, pp.45–46, Nara, Japan, 2002.

[17] S. Kawaguchi, P. K. Garg, M. Matsushita and K. Inoue, Automatic Categorization for Evolvable Software Archive, International Workshop on Principles of Software Evolution (IWPSE'03), pp.195–200, Helsinki, Finland, 2003.

[18] EASE (Empirical Approach to Software Engineering) project, http://www.empirical.jp/indexe.html

[19] J. D. Herbsleb, A. Mockus, T. A. Finholt and R. E. Grinter, An Empirical Study of Global Software Development: Distance and Speed, Proceedings of the 23rd international conference on Software engineering (ICSE'01), pp.81–90, Toronto, Canada, 2001.

[20] A. H. Dutoit and B. Bruegge, Communication Metrics for Software Development, IEEE Transactions on Software Engineering, Vol.24, No.8, pp.615–628, 1998.