

# Mega Software Engineering

Katsuro Inoue <sup>†,\*</sup>, Pankaj K. Garg <sup>‡</sup>,

Hajimu Iida<sup>††</sup>, Kenichi Matsumoto <sup>††,\*</sup>, Koji Torii <sup>††,\*</sup>

<sup>†</sup> Graduate School of Information Science and Technology, Osaka University  
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

<sup>‡</sup> Zee Source, 1684 Nightingale Avenue, Suite 201, Sunnyvale, CA 94087, USA

<sup>††</sup> Nara Institute of Science and Technology, Nara 630-0192, Japan

\* EASE (Empirical Approach to Software Engineering) Project, Senri, Osaka, Japan

inoue@ist.osaka-u.ac.jp, garg@zeesource.net

{iida, matumoto, torii}@is.aist-nara.ac.jp

## Abstract

*In various fields of computer science, rapidly growing hardware power, such as high-speed network, high-performance CPU, huge disk capacity, and large memory space, has been fruitfully harnessed, e.g., for large scale data and web mining, grid computing, and multimedia environments. We propose that such rich hardware can also catapult software engineering to the next level. Huge amounts of software engineering data can be collected from tens of thousands of projects inside organizations, or from outside an organization through the Internet. The collected data can be analyzed extensively to extract useful knowledge for improving organization-wide productivity and quality. We call such an approach for software engineering **Mega Software Engineering**. In this paper, we propose the concept of Mega Software Engineering, and demonstrate through several examples some of its core technologies. In addition, we propose an architectural framework for Mega Software Engineering.*

## 1. Introduction

### 1.1. Background and Motivation

Over the past few decades, software engineering has developed and introduced various technologies for software quality improvement and development efficiency. Various kinds of approaches for improvement have been proposed and accomplished so far, e.g., version control, configuration management, component reuse, and software process improvement, to name a few.

We believe that for the individual programmer or project

manager, however, these software engineering technologies remain focused on the individual project or programmer. For instance, code browsing tools typically allow a programmer to browse through single project code bases. Similarly, a navigation system might guide a developer utilizing data from her activities alone. While organizations can utilize global knowledge, for software reuse and other process improvements, the individual programmer or manager typically does not enjoy the benefits of *multi-project or global knowledge*.

Hence, prevailing organizational software engineering technologies for individuals are locally optimized to get local benefit for the individual developers or projects at most. They do not oversee global benefit and do not optimize the technologies using knowledge and software engineering data of other developers or other projects.

In modern times, the capacity, connectivity and performance of various networks ranging from local area network to the Internet are marvelously and rapidly growing. Huge numbers of computer systems are inter-connected using complex topologies, and various kinds of information on those systems can be instantly gathered. For example, Virtual Private Networks (VPN) has become pervasive in organizations in the past six months. Software data such as software process of individual developers or software products created by project is easily collected through networks. Now we are able to collect data from not only a single project, but *all* software development activities inside an organization (or company). If the organization has close relation to other software development organizations, as sub-contractor or co-developer, we can also collect software engineering data from the other organizations. Currently there is a huge collection of Open Source software on the Internet, and they are sometimes crucial resources

for development projects. They are easily searched and collected via Internet tools.

Disk capacity and CPU power of recent computer systems are astonishingly increasing. Since vast disk space is available, we can archive project data in very fine granularity. Every change of a product in a project can be recognized as a single version and be stored into a version control system. Every communication made between the developers can be recorded. Not only single project data, but also we can store and archive all project data spread over distributed organizations.

The collected software engineering data includes both process and product information of target projects. Various characteristics can be extracted by analyzing the collected data. Mining a single project data would be a relatively straightforward and light task. On the other hand, mining through multiple project data, say tens of thousands of projects, would be a complicated and heavy task. However, since now we have enormous computational power and memory space compared to, e.g., 10 years ago, such analysis becomes a feasible challenge. We may want to analyze, not only the organizational software engineering data, but also software engineering data available on the Internet as Open Source projects [33], such as various source programs, associated documents, version control logs, mail archives, and so on.

In computer science researches and practices, there are many successful uses of improved hardware capacity. For example, WEB data collection and mining such as Google search engine is a case in the WEB engineering field. In the high-performance computation field, GRID technology is an example. We think that the software engineering field should also share in advantage of the improvement of network, CPU, disk, and etc. We propose to create a novel approach in software engineering field, by collecting various software engineering data through networks extensively, archiving the collected data for a long period, analyzing the huge data deeply, and providing knowledge for organizational improvement. We call such network and CPU intensive approach for software engineering “Mega Software Engineering.”

Technologies in traditional software engineering tend to be based on limited knowledge in the sense of small data set of individual developer or project. Mega software engineering aims at global knowledge in the sense of huge data set of global projects. Even from existing software engineering approaches, there will be some technologies that fall into Mega Software Engineering, although many new methods and tools will come out soon if we clearly recognize the importance of Mega Software Engineering.

## 1.2 Paper Overview

In this paper, we will propose the concept of Mega Software Engineering and will discuss its feasibility and applicability.

First, we will show a classification of software engineering technologies by the scale of engineering target, and will depict distinction between Mega Software Engineering and traditional software engineering in Section 2.

Section 3 will introduce examples of core technologies of Mega Software Engineering. The first example is a Mega Software Engineering Environment, analogous to the environments utilized by tens of thousands of Open Source or Free software projects as exemplified by SourceForge [36] or Gnu Savannah [12]. For organizations, such environments are exemplified by Corporate Source [6, 7, 43]. They are composed of several development tools supporting open source development processes, with version control tools such as CVS [3], e-mail management tools such as Mailman [11], and bug tracking tools such as Gnats [10] or Bugzilla [28]. They also provide GUI's that control multiple projects and allow browsing from project to project. A project may be performed in widely distributed way in the globe, and the project data are gathered and archived at servers through private networks or the Internet. Currently, they provide limited features for deeply analyzing data collected at servers, but they will be important infrastructures for the data collection of Mega Software Engineering.

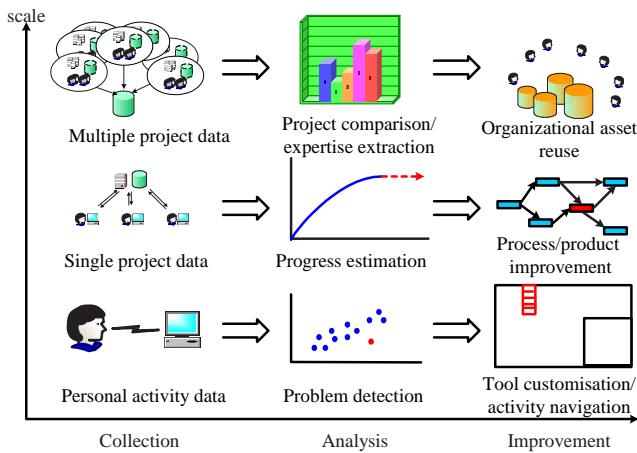
We will also show more analytical cases of software categorization, collaborative filtering, code-clone detection, and software component search.

We are currently developing a framework for investigating a wide range of Mega Software Engineering technologies. The framework is based on Mega Software Engineering Environment, with standardized databases and various analysis engines. Section 4 will discuss these.

Section 5 will discuss on our approach and will compare to related works. We will conclude our discussion in Section 6.

## 2. Overview of Mega Software Engineering

Figure 1 shows the classification of software engineering technologies based on the scale of engineering targets. The horizontal axis shows improvement feedback steps, composed of collection (measurement) step, analysis (evaluation) step, and feedback (improvement) step. The vertical axis represents the scale of the target for software engineering. We will explain each scale in the following.



**Figure 1. Scale classification of software engineering technologies**

### 2.1. Software Engineering for Individual Developer

The first scale level includes traditional software engineering technologies which target individual developers. Data and knowledge for each developer is collected and analyzed, then the result information is fed back to the individual developer.

For instance, command history of a tool for a developer is collected and analyzed so that arrangement of the tool menu can be improved or we can create a command navigation feature for the developer. Many software engineering tools such as software design tools, debug support tools, or communication support tools are in this category.

### 2.2. Software Engineering for Single Software Development Project

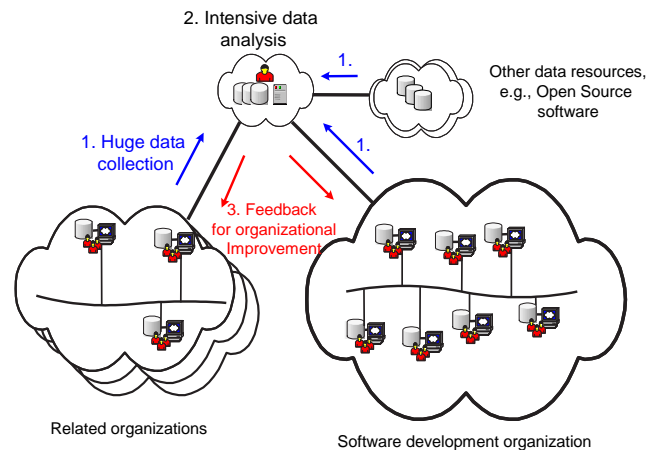
The second scale level includes also traditional software engineering technologies which target a single software development project, or a set of closely related development projects such as product-line development projects. The engineering data for the project is collected and analyzed so that improvement of the project's processes and products is established.

For example, we may collect product data such as the number of accomplished modules in a project, and then compare to the scheduled number. So we can know the current status of the project progress and we can improve the project process if needed. Process engineering tools and distributed development support tools are examples of this scale.

### 2.3. Software Engineering for Project Collection (Mega Software Engineering)

There have been little software engineering researches proposed and realized in this scale, since there had been limitations on network capacity, CPU power, and so on. Now those limitations have gone away; thus we can collect and analyze a large volume of data, and we can think about optimized strategies beyond individual or project boundaries. The results of the optimization would produce the benefit to software development organization, rather than the benefit to individual developers or projects

For example, we can gather multiple project data sets from overall organization, and can compare projects to projects to extract project natures. Analyzed data for project processes and products are archived as assets of the organization.



**Figure 2. Fundamental steps of Mega Software Engineering**

As shown in Figure 2, we consider that Mega Software Engineering is composed of following steps.

1. Huge data collection for a large number of projects
2. Intensive data analysis beyond boundary of projects
3. Information feedback for organizational improvement

Technologies in Mega Software Engineering relate to one of these three steps. We will show examples of the technologies in the following section.

### 3. Component Technologies of Mega Software Engineering

#### 3.1. Mega Software Engineering Environments

An essential component of Mega Software Engineering is the ability to collect and make available large amounts of data from tens of thousands of software projects. Rather than collect such data *a posteriori*, we propose that such data be collected as the software engineering work gets done. A critical aspect of this is to collect data as a *side-effect* rather than as an *after-thought*. This implies the existence of a **Mega Software Engineering Environment (MSEE)** that can easily accommodate the development effort of tens of thousands of projects. Fortunately, the Open Source and Free Software worlds have demonstrated the feasibility of such environments through the pioneering efforts such as SourceForge [36] and the Gnu software tools. In the rest of this section we will describe the architecture of one such MSEE, Corporate Source [43], with which we are most familiar. The other MSEE's (e.g., see [15]) have similar architecture.

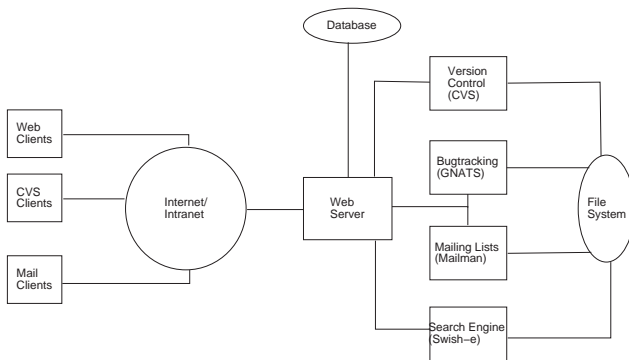


Figure 3. MSEE architecture

Figure 3 shows the main components of Corporate Source. As the figure shows, Corporate Source is a web-based service. Through the web interface, Corporate Source provides capabilities to:

- Add a new software project to the collection
- Browse through existing projects, using various sorting orders like categories, software name, contact name, or date of submission.
- Search through the software projects, either through the source code, software descriptions, mailing list archives, or issues and bug reports.

When a user adds a new software project, Corporate Source requires the user to input a set of information about

the software, e.g., who were the authors of the software, some keywords, a brief software description and title, etc. Corporate Source takes this information and stores it an XML file associated with the project. It also instantiates a version control repository, a mailing list, and a bug tracking system for that software project. Henceforth, users of Corporate Source can start working on the project using the version control repository for their source code management. As in the case of Open Source software, Corporate Source requires that all decisions making and discussions about the software project be carried out using the email discussion list associated with the project. In this manner, an archive will be maintained of the history of project decision making.

General users of the MSEE are free to browse through the source code and mailing list discussion forums to get a better understanding of the software. If they find any problems or issues with any software, they can input such issues in the bug tracking system associated with that software.

Hence, an MSEE provides some important features enabled by the rapid advances in network, CPU, and disk capacities:

- Maintain and make visible tens of thousands of software projects at the same time.
- Collect fine-grained data on each project for multiple versions, bug reports and their resolution, and feature design discussions.
- Provide a uniform web-based interface to all information.
- Collect data as side-effect of normal project activities.

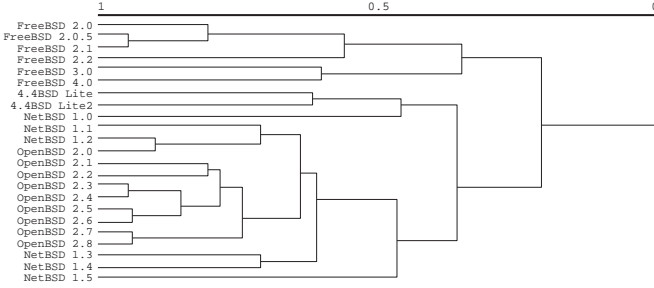
#### 3.2. System Categorization

MSEE provides a fundamental vehicle for collecting thousands of project data sets. From large project data stored in archives, we frequently want to search similar project data or related data; thus we need to know the similarity over projects or software systems made by projects.

It is unrealistic to categorize the number of projects by human hand. For example, SourceForge is a huge WEB site for Open Source software development projects, and it contains more than 67,000 projects at this moment. Categorization of each project into project groups is performed by human hand; however, we need deep expertise of not only the target project, but also categorization of projects.

We have been studying automatic categorization of software systems [22, 24, 41]. The first approach performs cluster analysis for the sets of source code [41]. This is based on the similarity of two sets of source code, which is defined as the ratio of the numbers of similar code lines to that of the overall lines of two software systems. The similar code

lines are detected by a combination of a code-clone detection tool CCFinder [21] mentioned below and a difference extraction tool *diff* [5].



**Figure 4. Dendrogram of BSD UNIX using source-code similarity**

Figure 4 is a dendrogram using the similarity as distance, for several dialects of BSD UNIX operating systems, i.e., 4.4BSD Lite, FreeBSD, NetBSD, and Open BSD. As you can see, each dialect is categorized very clearly, so we can visually identify the evolution of the BSD UNIX operating systems.

This approach is very effective in the cases that similar software systems such as ancestor or descendant versions are compared. On the other hand, it is not well applicable to the cases that target software systems share little number of source code lines, since the resulting similarity values are almost 0 and the difference of such values has almost no meaning.

For such cases, it seems that we would need categorization of software systems not by the shared source code lines, but by shared features or libraries used by the systems. Those features and libraries would be well reflected by analysis of the keywords involved in the source code.

We propose an approach of categorization of software systems using LSA (Latent Semantic Analysis) [25] for the keywords appearing in the source code of the target systems [22, 24]. LSA is a method for extracting and representing the contextual-usage meaning of words by statistical computations applied to a large corpus of text. It has been applied to a variety of uses ranging from understanding human cognition to data mining.

Table 1 shows the similarity values which is the cosine of the column vectors of the resulting matrix by LSA. We have chosen 11 software systems from SourceForge, and software groups D1–D3, E1–E3, and V1–V3, and X1–X2 are categorized by hand in the same groups at SourceForge. Two systems having 1 mean very similar, and those with 0 mean no similarity in the keyword lists.

Groups E, V, and X have very high similarities inside the groups. The result shows that although there are some outliers, it would give us a good intuition of categorization

	D1	D2	D3	E1	E2	E3	V1	V2	V3	X1	X2
D1:firebird-1.0.0.796	<b>1</b>	0.1	0.2	0	0	0	0.1	0.2	0	0	0
D2:mysql-3.23.49	0.1	<b>1</b>	0.1	0	0	0	0.1	0.2	0	0	0
D3:postgresql-7.2.1	0.2	0.1	<b>1</b>	0	0.1	0	0	<b>0.5</b>	0	0	0
E1:gnotepad+-1.3.3	0	0	0	<b>1</b>	<b>1</b>	<b>1</b>	0	0	0	0	0
E2:molasses-1.1.0	0	0	0.1	<b>1</b>	<b>1</b>	<b>1</b>	0	0.1	0	0	0
E3:peacock-0.4	0	0	0	<b>1</b>	<b>1</b>	<b>1</b>	0	0	0	0	0
V1:dv2jpg-1.1	0.1	0.1	0	0	0	0	<b>1</b>	<b>0.8</b>	<b>1</b>	0	0
V2:libcu30-1.0	0.2	0.2	<b>0.5</b>	0	0.1	0	<b>0.8</b>	<b>1</b>	<b>0.8</b>	0	0
V3:mjpgTools	0	0	0	0	0	0	<b>1</b>	<b>0.8</b>	<b>1</b>	0	0
X1:XTermR6.3	0	0	0	0	0	0	0	0	0	<b>1</b>	<b>1</b>
X2:XTermR6.4	0	0	0	0	0	0	0	0	0	<b>1</b>	<b>1</b>

**Table 1. Categorization of software systems by LSA**

of software groups. We further continue this approach to improve the categorization precision [23].

By adding such automated categorization tool as an analysis feature, managers and developers are easy to find similar projects or related projects to a target project, and they can obtain useful knowledge of past projects.

### 3.3. Selecting Similar Cases by Collaborative Filtering

In previous approach described above, we are able to know sets of software systems, which are very similar. However, we cannot specify which one system is the most similar one to a particular software system. Collaborative filtering can answer the most related one [34]. We have been studying it as a mean of identifying software features from activity data [30]. Here, we propose to apply the collaborative filtering technique to find a system (or project) from thousands of systems.

We assume that there is a list of  $\alpha$  metrics  $M = \{m_1, m_2, \dots, m_\alpha\}$  and a list of  $\beta$  systems  $P = \{p_1, p_2, \dots, p_\beta\}$ . Value  $v_{ij}$  can be obtained by applying metric  $m_i$  to the data set of system  $p_j$ . In similarity computation between two systems  $p_a$  and  $p_b$ , we first isolate the metrics, which had been applied to both of these systems, and then apply a similarity computation to the value of the isolated metrics. For example, two systems are thought of as two vectors in the  $\alpha$ -dimensional metric-space. The similarity between them is measured by computing the cosine of the angle between these two vectors. Once we can isolate the set of the most similar systems based on the similarity measures, we can estimate metric value  $v_{ij}$  even when a metric  $m_i$  is not available. In such case, an estimation value, such as a weighted average of the metric values of these similar systems, is employed.

This means that collaborative filtering is robust to the defective data sets. In contrast, the conventional regression analysis requires the complete matrix of metric values, and it is unrealistic to assume complete data sets for all systems.

If a project manager found deviation from the project plan in schedule, he/she has to do corrective action to bring expected future schedule performance in line with the project plan [32]. In such situation, the project manager may want to know a viable solution to the problem. Collaborative filtering can present a set of the most similar systems to the ongoing system, so that we can explore the product and process data collected in these similar systems, and we would find a concrete solution. We think that to proceed Mega Software Engineering effectively, we need to provide not only a bird's-eye view of software systems and projects, but also concrete information useful for software developers and project managers.

### 3.4. Code-Clone Detection

As an example of deep analysis for the large collection of software engineering data beyond project boundaries, we will show code-clone detection tools CCFinder and Gemini for large scale of source code [21, 40].

Code clone is a code fragment in a source file that is identical or similar to another. CCFinder takes a set of source-code files as an input, and generates a list of code-clone locations as the output. For efficient detection, CCFinder first performs a lexical analysis of the input source code, and obtains a single sequence of tokens. This token sequence is normalized and transformed to remove the effect of the difference of user-defined names or that of other meaningless clones. The resulting token sequence is analyzed to generate clone locations using the index tree algorithm [14]. The output of CCFinder is sent to Gemini to visualize distribution and characteristics of clones in the system.

Figure 5 is an example of the display of Gemini. This is the scatterplot of detected clones between two GUI libraries Qt [39] and GTK [13]. These two libraries are developed independently in different organizations. Qt (version 3.2.1) is composed of 929 files and about 686K lines in total. GTK (version 2.2.4) consists of 658 files and 546K lines in total.

Each dots in the scatterplot represents existence of code clones with more than 30 tokens. Smaller tokens less than 30 tokens are eliminated here. The left-upper pane shows clones inside Qt, and the right-lower pane shows clones inside GTK. The result is symmetrical to the main diagonal line, so the right-upper half is omitted.

The left-lower pane shows clones between Qt and GTK. The overall clone density in this pane is generally lower than others, but there is one exceptional portion annotated by "a", where there are many clones, meaning that two systems share most code. This portion is the font handler for both Qt and GTK, and we knew by reading readme files that the font handler of Qt is imported from GTK.

Using the code clone detection technique, we can determine similarity of source codes, leading to categorization of

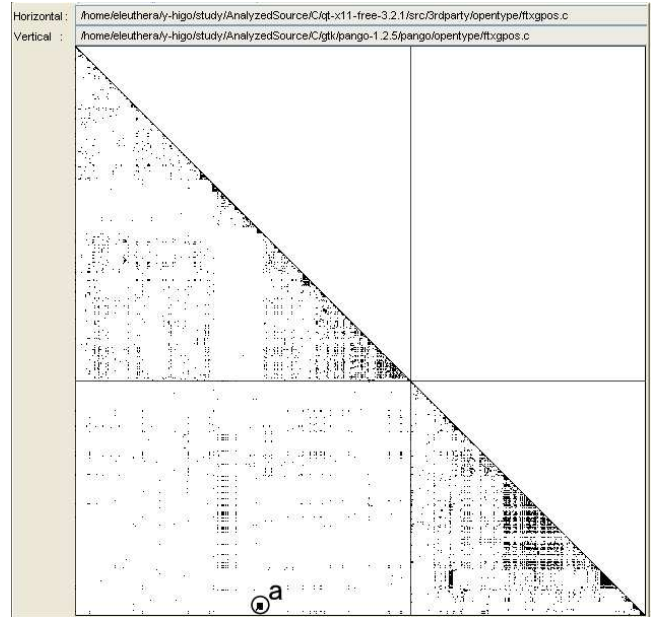


Figure 5. Scatterplot of clones between Qt and GTK

software systems. Also, we can create an effective search tool for similar code portion to the huge archive of organizational software assets.

The code clone detection requires high CPU power and huge memory space for million of source codes. For example, for two versions of Open Office [31], which are about 10M lines of code in total, CCFinder requires 68 minutes on Pentium IV 1.5GHz with 1 GByte memory. However, we think that they are affordable computer environment and analysis time.

### 3.5. Software Component Search

In the world of the Internet, or even inside a single organization, there would be many cases such that similar software components (code portions) are developed independently in different projects day by day.

Collecting software components and archiving them for reuse of the components are important issues. Constructing well-organized software libraries would be a very important objective in the organization; however, it requires a large amount of human resources if it would be developed by hand, and also it is very difficult to keep the libraries consistent and useful.

We have designed an automatic software component library that analyzes a large collection of software components, indexing them for efficient retrieval, and ranking them by the importance of components. We have proposed a novel method of ranking software components, called

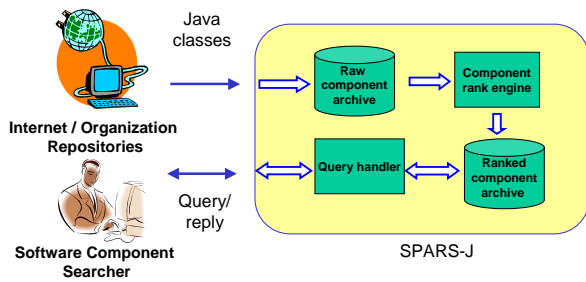


Figure 6. Architecture of SPARS-J

Component Rank, based on the analysis of actual use relations of components and also based on convergence of the significance values through the use relations [20].

The use relations among software components are represented by a directed graph, and the eigenvector with eigenvalue 1 for the adjacent matrix of the directed graph is computed. The sorted order by the values of each component in the eigenvector is the result rank for the component, and it shows its relative significance, i.e., more used components directly or indirectly by other components are ranked higher.

Using the component rank computation as a core ranking engine, we are currently developing Software Product Archiving, analyzing, and Retrieving System for Java, called *SPARS-J* [37]. Figure 6 shows *SPARS-J* architecture. Various Java source programs have been collected, and they are stored in the raw component archive. Each class in Java is considered as a component here. The collected components are ranked by the component rank engine and stored at the ranked component archive.

A component searcher, who is trying to build a software system, will give *SPARS-J* queries for some typical definition or typical usage of a class to build, by keywords possibly found in source code. These queries are analyzed at the query handler, and they are given to the ranked component archive. The keywords are searched through the archive, and the matched components are sorted by the ranks. The result component list is returned to the searcher through the query handler.

The archive currently contains more than 170,000 Java classes. It takes about one whole day for the component rank engine to parse, index, and rank all of them on a PC server with Pentium IV of 3GHz clock speed and 8 GByte memory.

Figure 7 shows a display result for a query keyword “bubblesort” for *SPARS-J*. The result is returned almost instantly to the searcher through a WEB browser.

There are 28 classes having the keyword. Similar or the same classes are merged into 19 groups out of 28 classes,



Figure 7. Query result of SPARS-J for “bubblesort”

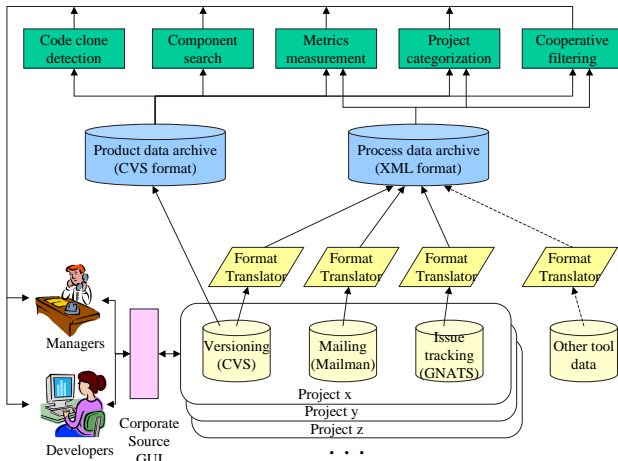
and these 19 groups are sorted by the component ranks. BubbleSort is the first ranked groups with two classes found from two different sites. The second group contains one class BubbleSortTest.

The details of listed classes, which include the source code, various metric values, and various links to other classes, can be viewed simply by clicking on the WEB browser.

This system would become a very powerful vehicle to manage organizational software assets. It is easy to collect all source code created in an organization at the raw component archive. Then, the analysis for the ranking and the retrieval for the query are performed fully automatically, without using human hand. So the cost of the software asset management would reduce drastically, and the developers can leverage past assets for efficient development and reliable products.

## 4. Implementation of Mega Software Engineering Framework

To investigate various technologies in Mega Software Engineering, we are currently developing a tool collection environment called *Mega Software Engineering Framework*, as shown in Figure 8. We do not intend to build a single huge system to perform all the steps in Mega Software Engineering, but we would build a framework to help to establish individual technologies in Mega Software Engineering.



**Figure 8. Architecture of Mega Software Engineering Framework**

This framework is composed of following three tool collections.

- Corporate Source as a Mega Software Engineering Environment, which manages project progress and collects project data
- Product and Process data archives
- Analysis tool set which extracts various feedback information

As described in Section 3.1, Corporate Source employs version management tool CVS, mail management tool Mailman, and issue (bug) tracking tool Gnats [10]. Corporate Source provides control and unified GUI for these tools; however, we can employ other tools for version control, mailing, or issue tracking. Note that the data collection by Corporate Source can be done non-intrusively. Checking into CVS repositories, sending mails, and tracking issues are performed as daily activities for software development and maintenance, not as special activities for the data collection.

As the central archives of this framework, we prepare a product data archive in the CVS format and a process data archive in an XML format. The product data archive directly reflects to the repositories of each project in the CVS format. The process data is obtained by transforming log files of CVS, Mailman, and Gnats into a standard format in XML, and it is stored into an XML database that is implemented by PostgreSQL with XML extension. This framework can easily handle process data obtained by other tools if the data is transformed into the standard format in XML.

The process data and product data in the archives are analyzed by a tool for measuring various metrics data and by the tools presented in previous sections. The analysis results are given back to the developers and managers. We will add some analysis tools here. Also, we are designing a unified GUI for analysis results, which would accomplish effective feedback to the developers and managers, leading to the organizational benefit.

## 5. Discussions

### 5.1. Distinction and Relation to Other Software Engineering Technologies

- Global Software Development

Due to the rapidly increasing network capacity and speed, and differentiated cost structures, Global Software Development is an active area of software engineering research and practice [17]. Although analysis shows deficiency of global software development, compared to same site work [16, 18], the importance of Global Software Development will increase from now on and strong support tools to ease site distance barrier are really needed. For example, Herbsleb and Mockus have proposed an “expertise browser” to help locate far flung experts and contributors of software modules [26].

The Mega Software Engineering framework provides a fundamental environment of code sharing and message exchanging for Global Software Development. Also, our approach provides directly needed knowledge or asset to developers or managers, rather than assistance of finding expertise.

- Knowledge Sharing

There are several researches in which light-weight knowledge is extracted and shared among developers [4, 42]. In [4], link information is analyzed and provided as related knowledge. In [42], a system automatically providing source-code components that is not well identified or understood by the developer is proposed.

The light weight approach focuses on a single developer or a single project. Our approach explores knowledge or

information which is based on deeper analyses of multiple projects and a huge collection of software engineering data.

#### - Measuring and Analyzing Open Source Project Data

A measurement tool collection for CVS and mail data has been proposed in [9]. It generates various statistical values for Open Source development projects. Also, CVS data is used for getting various process metric values in [8]. In [27], it analyzes CVS data to classify the causes of changes made to software products.

These approaches are also considered to be examples of analysis techniques in Mega Software Engineering. However, their systems are more specific to getting the objective statistical values or classification. We are trying to build a more flexible framework for a large collection of projects, in which we can extract both inter-project knowledge of process and product for various objectives. Thus, we employ an exchangeable standard format in XML for process data, and use a standard database to archive it. Once the data is in the form of a standard database, we can apply various techniques for data mining for traditional data.

#### - Measurement-Based Improvement Framework

There are a large number of researches and practices on frameworks of measurement and improvement. Goal Question Metrics paradigm is an example in which suitable metrics are derived from measurement objectives [1]. The frameworks of software process improvement such as CMM [35] and SPICE [38] are also cases that aim measurement-based improvement for organization, and Personal Software Process targets improvement for personal capability [19].

We might consider that Mega Software Engineering would be an improvement framework similar to those. However, Mega Software Engineering is different in the sense that it assumes organizational-wide huge data collection of many projects and software systems, rather than a single person or single project. Also, the analyses made by Mega Software Engineering are more intensive and deeper ones compared to per-project metric values made by some frameworks.

#### - Experience Factory

Vic Basili's group has developed and successfully applied the concept of an "Experience Factory," where organizations systematically collect and reuse past experiences [2]. Indeed, Neto et al. propose a "knowledge management" framework for storing such experience base for organizations [29]. We believe that Mega Software Engineering is an evolution of the Experience Factory concept, enriched from the "communal" aspects of Open Source software development. Hence, instead of requiring a separate organizational element that captures and packages relevant "experience elements," we propose to directly capture the contents

of software engineering activities, and make "experience" available through deep analysis of this raw data.

## 5.2. Benefit of Mega Software Engineering

The objectives and approaches of the feedback for organizational benefit would be generally vague and hard to formalize. In our examples shown in Section 3, we simply gave the analysis results as "organizational knowledge" back to the developers and managers. Even with such a simple feedback strategy, we would expect organizational benefits, such that the projects will be well controlled and the productivity will increase drastically due to the reuse of past product and process data. Also, we would expect to improve reliability of products using fault data of similar or related projects.

These improvements are heavily rely on the fact that we can fairly easily create organization-wide huge assets of past products and processes, which had not been well structured or managed by human hand or by traditional software engineering technologies.

## 5.3. Feasibility of Mega Software Engineering

We can say that Mega Software Engineering has been already achieved in some part as shown in Section 3. Data collection would be generally straightforward by using current technologies, such as Open Source development tools. Analyses for the collected data would be more difficult and would need more elaboration. However, current technologies for large-scale analysis on WEB, database, and source program will be good resources for the needed analyses. The goal of the feedback for the organizational benefit is sometimes unclear as mentioned above. Once a clear goal would be defined, we could design a more elaborative system to get an effective feedback.

We do not think that there is a single system which supports all the steps of the collection, analysis, and feedback in Mega Software Engineering. The system becomes so huge and it would not be well designed. We would prefer to have a flexible framework with various pluggable tools which can be replaced for the objectives and approaches of the data collection, analysis, and feedback.

## 6. Conclusions

We have proposed a novel concept of Mega Software Engineering, and presented several of its core technologies. Also, we have shown an architecture of the Mega Software Engineering framework that is currently under development.

Previous work in software engineering research has given limited attention to data collection and analysis of

tens of thousands of projects. Now we can have very powerful hardware at hand and we can apply various technologies to huge data collection and intensive analysis. Therefore, we believe that we are at the best starting point of Mega Software Engineering for organizational and communal benefit.

## References

- [1] V. R. Basili. *Goal Question Metrics Paradigm*, in *Encyclopedia of Software Engineering (J. Marciniak ed.)*, pages 528–532. John Wiley and Sons, 1994.
- [2] V. R. Basili and G. Caldiera. Improve Software Quality by Reusing Knowledge and Experience. *Sloan Management Review*, Fall:55–64, 1995.
- [3] B. Berliner. CVS II: Parallelizing Software Development. In *Winter USENIX Conference*, Washington, D.C., 1990.
- [4] D. Cubranic, R. Holmes, A. Ying, and G. C. Murphy. Tool for Light-weight Knowledge Sharing in Open-source Software Development. In *3rd WS Open Source SE*, pages 25–30, Portland, OR, USA, 2003.
- [5] Diffutls. <http://www.gnu.org/software/diffutls/>.
- [6] J. Dinkelacker and P. Garg. Corporate Source: Applying Open Source concepts to a corporate environment (Position Paper). In *1st WS Open Source SE*, Toronto, Canada, 2001.
- [7] J. Dinkelacker, P. Garg, D. Nelson, and R. Miller. Progressive Open Source. In *ICSE*, Orlando, Florida, 2002.
- [8] D. Draheim and L. Pekacki. Process-Centric Analytical Processing of Version Control Data. In *Int. WS Principles of Software Evolution*, pages 131–136, Helsinki, Finland, 2003.
- [9] D. German and A. Mockus. Automating the Measurement of Open Source Projects. In *3rd WS Open Source SE*, pages 63–68, Portland, OR, 2003.
- [10] Gnu. Gnats Project. <http://www.gnu.org/software/gnats>.
- [11] Gnu. Mailman Project. <http://www.lists.org>.
- [12] Gnu. Savannah Project. <http://savannah.gnu.org>.
- [13] GTK Project. <http://www.gtk.org>.
- [14] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*, pages 89–180. Cambridge University Press, Mass., 1997.
- [15] T. J. Halloran, W. L. Scherlis, and J. R. Erenkrantz. Beyond Code: Content Management and the Open Source Development Portal. In *3rd WS Open Source SE*, Portland, OR, USA, 2003.
- [16] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An Empirical Study of Global Software Development: Distance and Speed. In *ICSE*, pages 81–90, Toronto, Canada, 2001.
- [17] J. D. Herbsleb and D. Moitra. Global Software Development. *IEEE Software*, 18(2):16–20, 2001.
- [18] J. D. Herbsleb and D. Moitra. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE TSE*, 29(6):481–494, 2003.
- [19] W. S. Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley, 1996.
- [20] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component Rank: Relative Significance Rank for Software Component Search. In *ICSE*, pages 14–24, Portland, OR, 2003.
- [21] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE TSE*, 28(7):654–670, 2002.
- [22] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic Categorization for Evolvable Software Archive. In *Int. WS Principles of Software Evolution*, pages 195–200, Helsinki, Finland, 2003.
- [23] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic Categorization Tool for Open Software Repositories. In *WS Open-Source in an Industrial Context*, Anaheim, CA, 2003.
- [24] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. On Automatic Categorization of Open Source Software. In *3rd WS on Open Source SE*, pages 79–83, Portland, OR, 2003.
- [25] T. K. Landauer, P. W. Foltz, and D. Laham. Introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.
- [26] A. Mockus and J. D. Herbsleb. Expertise Brower: A Quantitative Approach to Identifying Expertise. In *ICSE*, pages 503–512, Orlando, FL, 2002.
- [27] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes Using Historic Database. In *ICSM*, pages 120–130, San Jose, CA, 2000.
- [28] Mozilla. Bugzilla Project. <http://www.bugzilla.org>.
- [29] M. G. M. Neto, C. B. Seaman, V. Basili, and Y. Kim. A Prototype Experience Management System for a Software Consulting Organization. In *SEKE 2001*, Buenos Aires, Argentina, June 2001.
- [30] N. Ohsugi, A. Monden, and S. Morisaki. Collaborative Filtering Approach for Software Function Discovery. In *Int. Symp. Empirical SE (ISESE)*, vol.2, pages 45–46, Nara, Japan, 2002.
- [31] Open Office Project. “<http://www.openoffice.org>”.
- [32] Project Management Institute, A Guide to the Project Management Body of Knowledge 2000 Edition, 2000.
- [33] E. S. Raymond. *The Cathedral and the Bazaar*. O’Reilly, 1999.
- [34] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based Collaborative Filtering Recommendation Algorithms. In *Int. World Wide Web Conf. (WWW10)*, pages 285–295, Hong Kong, 2001.
- [35] Software Engineering Institute, Carnegie Mellon University, <http://www.sei.cmu.edu/cmm/>.
- [36] SOURCEFORGE.net. <http://sourceforge.net/>.
- [37] SPARS Project, Osaka University Software Engineering Lab. <http://www.spars.info/SPARS/index.html.en>.
- [38] SPICE Project. <http://www.sqi.gu.edu.au/spice/>.
- [39] Trolltech. Qt. <http://www.trolltech.com>.
- [40] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code Clone Analysis Tool. In *Int. Symp. Empirical SE (ISESE)*, vol.2, pages 31–32, Nara, Japan, 2002.
- [41] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. Technical Report of Dept. of ICS, Osaka University, IIP-03-03-02, 2002.
- [42] Y. Ye and G. Fischer. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *ICSE*, pages 513–523, Orlando, FL, 2002.
- [43] ZeeSource. Corporate Source. <http://www.zeesource.net/>.