Mega Software Engineering

Katsuro Inoue[1], Pankaj K. Garg[2], Hajimu Iida[3], Kenichi Matsumoto[3], and Koji Torii[3]

[1] Osaka University, Graduate School of Information Science and Technology, 1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
inoue@ist.osaka-u.ac.jp
[2] Zee Source, 1684 Nightingale Avenue, Suite 201, Sunnyvale, CA 94087, USA
garg@zeesource.net
[3] Nara Institute of Science and Technology, Nara 630-0192, Japan
{iida, matumoto, torii}@is.naist.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. Examples of such usage are 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 systematically collected and organized 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 and correlate multi-project 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 some novel data analysis characteristic of Mega Software Engineering. We describe a framework for enabling Mega Software Engineering.

1 Introduction

Over the years, sometimes borrowing from traditional engineering disciplines, software engineering has adopted several methods and tools for developing software products, or more recently, software product families. For example, from hardware engineering the concept of specifying requirements before design and implementation have been useful for software engineering. A unique feature of software products, however, is that the end product has virtually no physical manifestation. Hence, composing or taking apart a software product has virtually no cost implications. As a result, software component reuse is a common practice for code sharing among multiple projects.

We posit that "sharing" among software projects can be extended beyond code or component sharing to more and varied kinds of "knowledge" sharing. Such sharing can be achieved using what we call *mega software engineering.* Instead of narrowly engineering a product, or a product family, an organization can undertake the responsibility and benefits of engineering a large number of projects simultaneously. Examples of benefits that can accrue from such a perspective are: projects that share functionality can benefit from code sharing or reuse; experts in a particular implementation aspect can contribute their expertise to all projects that can potentially use that expertise (sort of like syndicated newspaper columnist or cartoonists); historical experiences of projects can be extrapolated to similar, newer projects to eliminate repeating process mistakes; and, 'outliers,' or projects with behavior deviant from the norm can be easily distinguished for rapid problem identification and resolution.

Many existing 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, an individual programmer or manager seldom enjoys the benefits of *mega or global knowledge.* Often, in large organizations its difficult for programmers to even discover projects related or similar to their own.

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 growing rapidly. 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. A huge collection of Open Source software now exists on the Internet, which is sometimes a crucial resource for development projects. Such information is readily available via Internet tools.

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

The collected mega software engineering data includes both process and product information. 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 mega data, say tens of thousands of projects, can be computationally expensive. Since now we have enormous computational power and memory space compared to, e.g., 10 years ago, however, such analysis becomes feasible. 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, such as various source programs, associated documents, version control logs, mail archives, and so on.

In computer science research and practice, 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, etc. We propose to create a novel approach to software engineering field, by collecting mega software engineering data through networks, archiving the collected data for a long period, analyzing the huge data deeply, and providing knowledge for organizational improvement.

Undertaking mega engineering, however, is not straightforward. In addition to changing the programmer's mindset from engineering one product to multiple products, one has to accommodate the complexities of challenging the hierarchical socio-organizational context in which single product engineering is so deeply embedded. In this paper, we do not attempt to address such socio-organizational aspects, which have been addressed elsewhere[1]. Here we focus our efforts in describing the technology aspects of *Mega Software Engineering*: the novel analyzes enabled by performing data analysis on multiple projects, the architecture of a Mega software engineering environment, and a framework for collecting analysis data from the environment.

We depict the distinction between Mega Software Engineering and traditional software engineering in Section 2. In Section 3 we introduce some examples of core technologies of Mega Software Engineering. Section 4 outlines the framework based on Mega Software Engineering Environment. In Section 5 we compare this work to some related work, and conclude our discussion in Section 6 with a summary.

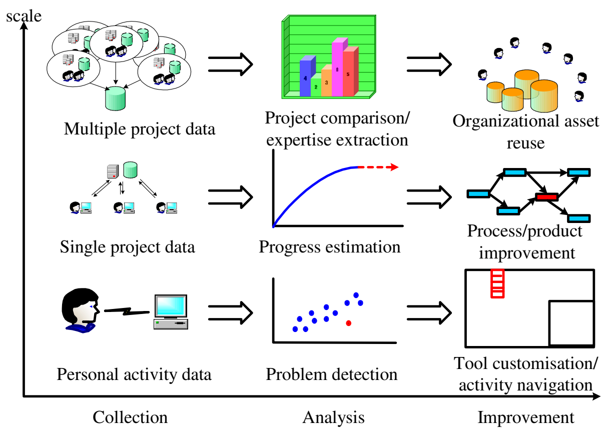

scale
Multiple project data
Project comparison/ expertise extraction
Organizational asset reuse
Single project data
Progress estimation
Process/product improvement
Personal activity data
Problem detection
Tool customisation/ activity navigation
Collection
Analysis
Improvement

Fig. 1. Scale classification of SE

2 Overview of Mega Software Engineering

Figure 1 shows a 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, which we explain in the rest of this section.

Individual Developer Software Engineering: 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 resulting analysis is fed back to the individual developer. For instance, command history of a tool for a developer can be collected and analyzed to improve the arrangement of the tool's menu bar, or to create a command navigation feature for the developer. Many software engineering tools such as software design tools, debug support tools, or communication support tools fall in this category.

Single Project Software Engineering: The *second scale level* includes current 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 to improve the project's processes and products. For example, we may collect product data such as the number of completed modules in a project, and then compare to the scheduled number. Such data can be used to monitor the project's progress and corrective action can be taken as necessary. Process engineering tools and distributed development support tools are examples of the Single Project Software Engineering scale.

Mega Software Engineering: At the *ultimate scale level*, we gather multiple project data sets from the entire organization, and compare among projects to draw meaningful conclusions. Analyzed data for project processes and products can be archived as assets of the organization. We note that there have been little software engineering research proposed and realized at this scale, since traditionally there has been limitations on network capacity, CPU power, and so on. Now those limitations have gone away; we can collect and analyze a large volume of data, and we can consider optimization strategies beyond individual or project boundaries. The results of such optimization will benefit the entire software development organization and its members, rather than benefiting simply a single developer or project.

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

1. huge data collection for a large number of projects,
2. intensive data analysis beyond boundary of projects, and
3. information feedback for organizational improvement.

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

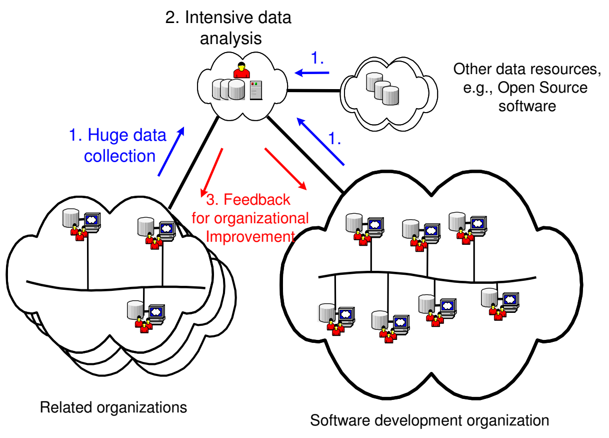

2. Intensive data analysis
1.
Other data resources, e.g., Open Source software
1. Huge data collection
1.
3. Feedback for organizational Improvement
Related organizations
Software development organization

Fig. 2. Fundamental steps of Mega Software Engineering

3 Component Technologies of Mega Software Engineering

3.1 Mega Software Engineering Environment

An essential component of Mega Software Engineering is the ability to systematically collect and organize large amounts of data, from tens of thousands of software projects. This requires: (1) mechanisms for defining the data to be collected from each project, (2) systematic organization of the collected data, and (3) mechanisms for easily obtaining the data from each project.

For each of these questions, we learn from the experiences of the Open Source and Free Software communities that have demonstrated an environment for collecting and organizing vast amounts of mega data, through the pioneering efforts such as Open Source Development Network (OSDN) and the Gnu software tools. Hence, similar to the OSDN, for each project we capture complete versioned source code trees, email discussion archives, bug report and their workflow, and documents associated with the project including web pages. We use the a combination of the hierarchical file system and relational database to organize the large amounts of data.

Rather than collect such data *a posteriori*, we collect and organize such data *in situ*. 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. In the following, we briefly describe the architecture of one such MSEE, SourceShare [2][3], with which we are most familiar. Other MSEE's (e.g., see [4]) have similar architecture.

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

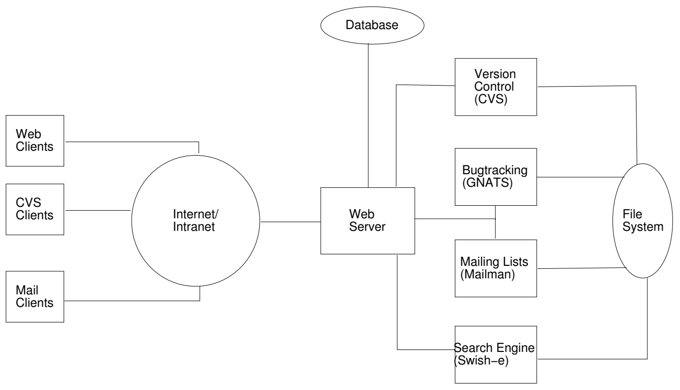


Fig. 3. MSEE architecture

- 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, SourceShare 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. SourceShare stores this information in 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 SourceShare can start working on the project using the version control repository for their source code management. As in the case of Open Source software, SourceShare requires that all decision making and discussions about the software project be carried out using the email discussion list associated with the project. In this manner, SourceShare maintains an archive of the history of project decision making.

An MSEE provides some important features:

- Maintain and make visible tens of thousands of software projects.
- Systematically collect and organize fine-grained data on each project for source code versions, problem reports and their resolution, and project discussions.
- Provide a uniform web-based interface to all information.
- Collect data as side-effect of normal project activities.

3.2 Automatic Categorization

MSEE provides a fundamental vehicle for collecting thousands of project data sets. Within the large project data stored in archives, users frequently want to find clusters of “similar” projects. Hence, we need mechanisms to determine related projects in a large corpus of multi-projects.

In the Open Source and Free Software communities, categorization is carried out by human input, usually at the beginning of the project. It is unrealistic, however, to consider human categorization given the multitude of software systems that can be expected in a typical mega environment. For example, SourceForge is a huge web site for Open Source software development projects, and as of this writing it contains about 78,000 projects. Human categorization would require not only a good understanding of the individual project to be categorized, but the potential categories that can be created by upto 78,000 projects.

To this end, we are studying automatic categorization of software systems [5][6]. The first approach performs cluster analysis for the sets of source code [6]. 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 [7] and a difference extraction tool *diff*.

For categorization of software systems with little shared code, we propose another approach of categorization of software systems using LSA (Latent Semantic Analysis) [8] for keywords appearing in the source code of the target systems [5]. 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.

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

Table 1. Categorization 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. Table 1 shows the similarity values which are the cosines of the column vectors of the resulting matrix by LSA. Two systems having a 1 entry implies they are 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 of software groups. We further continue this approach to improve the categorization precision.

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

3.3 Selecting Similar Cases by Collaborative Filtering

In the approach described above, we are able to identify cluster of software systems that are similar to each other. We cannot, however, specify which one system is the most similar to any given software system. Collaborative filtering can answer this question of finding the project most related to a given system [9]. We are studying such collaborative filtering as a means of identifying software features from activity data [10]. Here, we propose to apply the collaborative filtering technique to find a similar system (or project) from thousands of systems.

We assume that there is a list of α metrics $M = \{m_1, m_2, \ldots, m_\alpha\}$ and a list of β systems $P = \{p_1, p_2, \ldots, p_\beta\}$. 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 α-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 finds a deviation from the scheduled project plan, she has to take corrective action to bring future performance in line with the project plan [11]. In such a situation, the project manager may want to know a viable solution for 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 find a concrete solution. Hence, to proceed with 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 tool CCFinder and its GUI Gemini for large scale of source code [7].

Code clone is a code fragment in a source file that is identical or similar to another fragment. CCFinder takes a set of source-code files as an input, and generates a list of code-clone locations as the output.

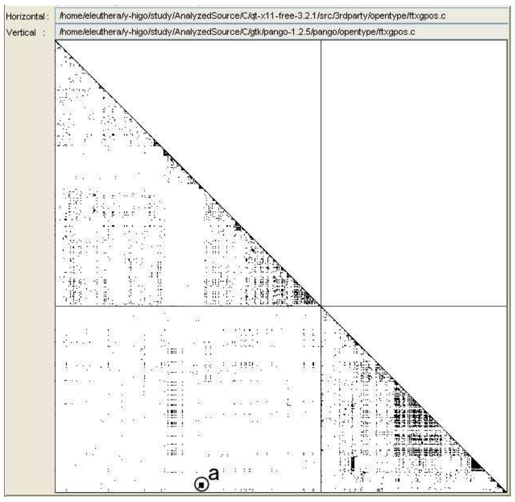


Fig. 4. Scatterplot between Qt and GTK

Figure 4 is an example of the display of Gemini. This is the scatterplot of detected clones between two GUI libraries Qt and GTK. 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 insider GTK. The result is symmetrical to the main diagonal line, so the right-upper half is omitted.

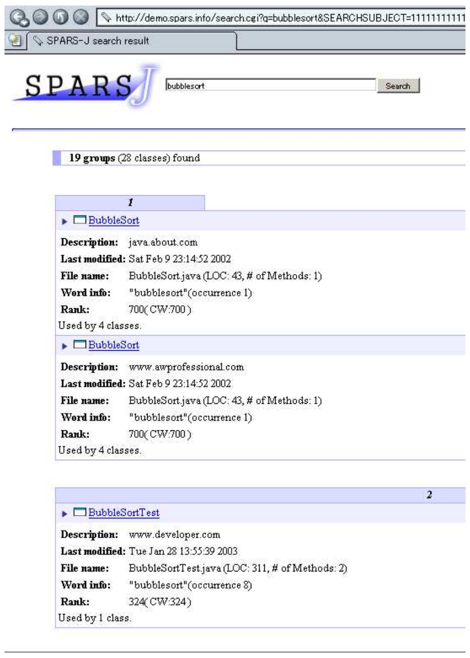
The left-lower pane shows clones between Qt and GTK. The overall clone density in this pane is generally lower than others, but there is one exceptional portion annotated by "a", where there are many clones, meaning that two systems share most code. This portion is the font handler for both Qt and GTK,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

Using these tools, we can quantify similarity of source codes, leading to categorization of software systems and to measurement of code reuse. Also, we can create an effective search tool for similar code portion to the huge archive of organizational software assets.

3.5 Software Component Search

Automation of reusable software component libraries is an important issue in organization. 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 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 [12].

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*.


http://demo.spars.info/search.cgi?q=bubblesort&SEARCHSUBJECT=11111111111

SPARS-J search result

bubblesort

19 groups (28 classes) found

1

BubbleSort

Description: java.about.com
Last modified: Sat Feb 9 23:14:52 2002
File name: BubbleSort.java (LOC: 43, # of Methods: 1)
Word info: "bubblesort"(occurrence 1)
Rank: 700(CW:700)
Used by 4 classes.

BubbleSort

Description: www.awprofessional.com
Last modified: Sat Feb 9 23:14:52 2002
File name: BubbleSort.java (LOC: 43, # of Methods: 1)
Word info: "bubblesort"(occurrence 1)
Rank: 700(CW:700)
Used by 4 classes.

2

BubbleSortTest

Description: www.developer.com
Last modified: Tue Jan 28 13:55:39 2003
File name: BubbleSortTest.java (LOC: 311, # of Methods: 2)
Word info: "bubblesort"(occurrence 8)
Rank: 324(CW:324)
Used by 1 class.

Fig. 5. SPARS-J for "bubblesort"

Figure 5 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, and these 19 groups are sorted by the component ranks. 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 can become a very powerful vehicle to manage organizational mega 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 can reduce drastically, and developers can leverage past assets for efficient development of reliable products.

4 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 6. We do not intend to build a single huge system to perform all the steps in Mega Software Engineering, but we construct a pluggable framework in which individual technologies for Mega Software Engineering can easily be incorporated.

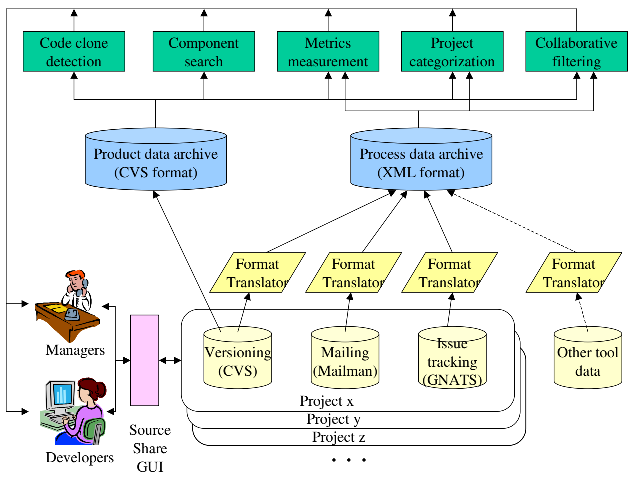


Fig. 6. Architecture of Mega Software Engineering Framework

This framework is composed of following three tool collections: (1) SourceShare as a Mega Software Engineering Environment, which manages project progress and collects project data, (2) Product and Process data archives, and (3) analysis tools which extract various feedback information.

As described in Section 3.1, SourceShare employs version management tool CVS, mail management tool Mailman, and issue (bug) tracking tool Gnats [13]. SourceShare 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 SourceShare is 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 are designing a unified GUI for analysis results, which would accomplish effective feedback to developers and managers.

Now we briefly show an example scenario of using the mega software engineering framework.

Step 1: Progress Monitoring under the Framework
A project, say Project X, starts with the mega software engineering framework. The progress is monitored by the metrics measurement tool in the framework, and the current metric values are compared with scheduled metrics values, so that the current status of the project is recognized. Now we assume that the project is behind schedule.

Step 2: Similar Project Search
In order to investigate the cause of the problem on Project X, we find projects similar to X. This is performed with the project categorization tool and collaborative filtering tool. The project categorization tool, based on the automatic categorization method, provides a set of similar projects to X. The collaborative filter chooses a past project Y as the most similar from the "similar" set.

Step 3: Investigation by Various Metric Views
The evolution of various metric values of Y through its beginning to end is investigated, and compared to the metrics values of X. Some outlier metric values are identified. Assume here that X's metric value for reuse rate of software components is much worse than that of Y. This can be detected by measuring amount of code clones between Project X and past projects including Y.

Step 4: Improvement
Since we know that the reuse in Project X is not as actively done as Project Y, we try to promote the reuse actively in X by using the software component search tool. The developers start to browse the product data archive and to search useful legacy software components in Y or similar projects. The reuse rate of X increases to the similar level of Y, and the schedule delay is recovered.

5 Related Work

- 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 [14]. Although analysis shows deficiency of global software development, compared to same site work [15], the importance of Global

Software Development will continue to increase and strong support tools to ease site distance barrier are required. For example, Herbsleb and Mockus have proposed an “expertise browser” to help locate far flung experts and contributors of software modules [16]. 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 a point solution of e.g., providing assistance in finding expertise.

- Knowledge Sharing
  There are several researchers investigating light-weight knowledge extraction and sharing among developers. For example, Curanic *et al.* analyze link information to provide related knowledge [17], while Ye and Fischer describe a system for automatically providing source-code components that is not well identified or understood by a developer [18]. 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
  German and Mockus have proposed a measurement tool collection for CVS and mail data [19]. They generate various statistical values for Open Source development projects. Similarly, Draheim and Pekacki use CVS data to determine several process metric values [20]. Finally, Mockus and Votta use CVS data to classify the causes of changes made to software products [21]. 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 is a large body of research and practical implementation of measurement and improvement frameworks. Goal Question Metrics paradigm is an example in which suitable metrics are derived from measurement objectives [22]. The frameworks of software process improvement such as CMM and SPICE are also cases that aim measurement-based improvement for organization, and Personal Software Process targets improvement for personal capability [23]. 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 earlier frameworks.
  Also, Mega Software Engineering can be considered as a framework in the context of software process improvement such as CMMI or ISO-9001. Applicability and

effectiveness of Mega Software Engineering in such context have to be explored further in various industrial environments.

- 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 [24]. Indeed, Neto et al. propose a "knowledge management" framework for storing such experience base for organizations [25]. Dingsoyr et al. report practical experiences and recommendations for "knowledge reuse" [26]. 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.

6 Summary

We have proposed a novel concept of Mega Software Engineering, and presented several of its core technologies. We described a framework that allows for pluggable technologies for mega software engineering. Previous work in software engineering research has given limited attention to data collection and analysis of tens of thousands of projects. Rapid advances in hardware and communication technologies allow the application of various technologies to huge data collection and intensive analysis. Therefore, we believe that we are at the best starting point to utilize the benefits of Mega Software Engineering.

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

References

1. Melian, C., Ammirati, C., Garg, P.K., Sevon, G.: Collaboration and Openness in Large Corporate Software Devlopment. In: Presented at the European Academy of Management Conferenec, Stockholm, Sweden (2002) Available from http://www.zeesource.net/kc.shtml.
2. Dinkelacker, J., Garg, P., Nelson, D., Miller, R.: Progressive Open Source. In: ICSE, Orlando, Florida (2002)
3. ZeeSource: (SourceShare) http://www.zeesource.net.
4. Halloran, T.J., Scherlis, W.L., Erenkrantz, J.R.: Beyond Code: Content Management and the Open Source Development Portal. In: 3rd WS Open Source SE, Portland, OR, USA (2003)
5. Kawaguchi, S., Garg, P.K., Matsushita, M., Inoue, K.: Automatic Categorization for Evolvable Software Archive. In: Int. WS Principles of Software Evolution, Helsinki, Finland (2003) 195–200
6. Yamamoto, T., Matsusita, M., Kamiya, T., Inoue, K.: Measuring Similarity of Large Software Systems Based on Source Code Correspondence. In: PROFES 2005, Oulu Finland (2005)

7. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE TSE **28** (2002) 654–670
8. Landauer, T.K., Foltz, P.W., Laham, D.: Introduction to latent semantic analysis. Discourse Processes **25** (1998) 259–284
9. Sarwar, B., Karypis, G., Konstan, J., Riedl, J.: Item-based Collaborative Filtering Recommendation Algorithms. In: Int. World Wide Web Conf. (WWW10), Hong Kong (2001) 285–295
10. Ohsugi, N., Monden, A., Morisaki, S.: Collaborative Filtering Approach for Software Function Discovery. In: Int. Symp. Empirical SE (ISESE), vol.2, Nara, Japan (2002) 45–46
11. : Project Management Institute, A Guide to the Project Management Body of Knowledge 2000 Edition (2000)
12. Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S.: Component Rank: Relative Significance Rank for Software Component Search. In: ICSE, Portland, OR (2003) 14–24
13. Gnu: (Gnats Project) http://www.gnu.org/software/gnats.
14. Herbsleb, J.D., Moitra, D.: Global Software Development. IEEE Software **18** (2001) 16–20
15. Herbsleb, J.D., Moitra, D.: An Empirical Study of Speed and Communication in Globally Distributed Software Development. IEEE TSE **29** (2003) 481–494
16. Mockus, A., Herbsleb, J.D.: Expertise Brower: A Quantitative Approach to Identifying Expertise. In: ICSE, Orlando, FL (2002) 503–512
17. Cubranic, D., Holmes, R., Ying, A., Murphy, G.C.: Tool for Light-weight Knowledge Sharing in Open-source Software Development. In: 3rd WS Open Source SE, Portland, OR, USA (2003) 25–30
18. Ye, Y., Fischer, G.: Supporting Reuse by Delivering Task-Relevant and Personalized Information. In: ICSE, Orlando, FL (2002) 513–523
19. German, D., Mockus, A.: Automating the Measurement of Open Source Projects. In: 3rd WS Open Source SE, Portland, OR (2003) 63–68
20. Draheim, D., Pekacki, L.: Process-Centric Analytical Processing of Version Control Data. In: Int. WS Principles of Software Evolution, Helsinki, Finland (2003) 131–136
21. Mockus, A., Votta, L.G.: Identifying Reasons for Software Changes Using Historic Database. In: ICSM, San Jose, CA (2000) 120–130
22. Basili, V.R. In: Goal Question Metrics Paradigm, in Encyclopedia of Software Engineering (J. Marciniak ed.). John Weily and Sons (1994) 528–532
23. Humphrey, W.S.: Introduction to the Personal Software Process. Addison-Wesley (1996)
24. Basili, V.R., Caldiera, G.: Improve Software Quality by Reusing Knowledge and Experience. Sloan Management Review **Fall** (1995) 55–64
25. Neto, M.G.M., Seaman, C.B., Basili, V., Kim, Y.: A Prototype Experience Management System for a Software Consulting Organization. In: SEKE 2001, Buenos Aires, Argentina (2001)
26. Conradi, R., Dingsoyr, T.: Software Experience Bases: A Consolidated Evaluation and Status Report. In: 2nd PROFES 2000, Oulu, Finland (2000) 391–406