Multi-Project Software Engineering: An Example

Pankaj K Garg
garg@zeesource.net
Zee Source
1684 Nightingale Avenue, Suite 201, Sunnyvale, CA 94087, USA

Thomas Gschwind
tom@infosys.tuwien.ac.at
Technische Universität Wien
Argentinierstraße 8/E1841, A-1040 Wien, Austria

Katsuro Inoue
inoue@ist.osaka-u.ac.jp
Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

Abstract

In this paper we present an approach for developers to benefit from multi-project software knowledge. As we show in this paper, this can be achieved by gathering information about how numerous software projects are being built, and about the interrelation of the modules within the projects. Compared to approaches that only monitor a single project, the contribution of our approach is that it not only supports the reuse of isolated software modules or libraries but also the knowledge surrounding the code and individual projects. For instance, if a component is replaced with another probably better implementation within a project, this knowledge can be shared with **all** *relevant projects. In this paper, we show how the collection of such data allows developers to learn about such decisions from other projects, and hence how to benefit from such "multi-project" knowledge.*

1. Introduction

Recent advances in computer and networking hardware have enabled the collection and analysis of huge amounts of information. In this paper we present the idea that such advances can be leveraged for *Multi-Project Software Engineering*, i.e., engineering of thousands of software projects simultaneously. In the past, software reuse has focused on sharing code among projects, whether through black-box or white-box reuse. With Multi-Project Software Engineering, not only does code get reused across projects, but *knowledge* surrounding the code and the project gets extensively reused among the projects.

For example, if there's a break-down in a reusable component in one project, then information about the problem can be instantaneously broadcast to the thousands of users of that component. Correspondingly, each of the other projects does not have to independently discover the problem, and waste time through redundant problem resolution processes. Similar mechanisms can be put in place to avoid extra wasted effort when temporary or permanent fixes are discovered for the component's problem.

Enabling such Multi-Project Software Engineering requires a networked infrastructure that manages process and product information for multiple projects effectively. Design choices of what information gets stored, and how, will have substantial impact on the functionality and power of the resulting engineering processes. To that effect, we borrow heavily from the lessons learned from decades of working of the Open Source communities [14]. As such, we describe an architecture of storing and utilizing Multi-Project Software Engineering data, leveraging some of the key technologies developed by the Open Source community for supporting their own software development processes.

We are actively researching different methods of analyzing the vast amounts of multi-project software engineering data. For example, one promising area of work is automatic categorization of software systems, based on the source code of the systems, or keywords and comments associated with the source code [9, 10, 11]. In this paper, we describe another significant analysis opportunity: utilizing multi-project data to improve the effectiveness of the reuse processes for component-based reuse. In particular, we describe the opportunity for real-time and continuous adaptation of the "best possible" component for a multitude of projects.

Suppose a component C is used by a multitude of software systems S_1 through S_n. Further, suppose that there are m implementations of C, C_1 through C_m, and each implementation has a slight variation in the interface provided, operating system supported, performance characteristic, and so forth. Hence, each of the n projects have to make a choice about which implementation of the component to use. Moreover, as time progresses, and the implementations of the components change, the projects have to keep making these choices. Current technologies do not provide much support for one project's choices influencing other projects, except through out-of-band communication or coordination among the projects.

Through Multi-Project Software Engineering, we can provide automation for several aspects of this reuse process: (1) we cluster related components together, e.g., by their usability or interface provided, (2) we rank the components by their popularity among S_1 through S_n, and (3) we can provide automatic substitution of components if and when a problem is discovered in one of the implementations. In the rest of this paper we give an overview of technologies used to provide such automation.

2. Storing and Utilizing Data

An essential component of Multi-Project 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 environments for collecting and organizing vast amounts of multi-project data, through the pioneering efforts such as the Open Source Development Network (OSDN) [15] 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 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 Multi-Project 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 [1, 2], with which we are most familiar. Other MSEE's (e.g., see [7]) have similar architecture.

SourceShare is a web-based service. Through the web interface, SourceShare 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, 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, thereby maintaining a history of project decision making.

General users of SourceShare 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,
- 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, and
- collect data as side-effect of normal project activities.

3. Multi-Project Analysis

In a multi-project environment, larger projects typically rely on components implemented as part of other, probably smaller projects. A potential problem in such a work environment, especially in an Open Source setting, is that projects may die and are no longer maintained by their developers. In the subsequent discussion, we use project D to denote such a project. If such a situation arises, projects that rely on a component implemented as part of project D need to find other components providing the same or a similar functionality.

Typically, if such a situation arises, the maintainers of all the projects relying on a component that was provided by project D are seeking for alternatives. Thus, the process of locating projects that provide a viable alternative is duplicated several times. Using multi-project analysis, it is possible to base ones decision on the decision made by other projects that have used the same project D in the past. That is, one can issue queries such as "*which other projects have been using components provided by project D, and which other projects did they use in order to substitute the components originally provided from D?*"

One challenge is that for different projects, different substitute projects may be adequate, hence we must rank the components according to their popularity or similarity to the original project D. In order to solve this problem, we can use the component rank system that we have presented in [8].

Another challenge is that the substitute component typically provides a different interface than that provided by the original component D. In this case, it is necessary to adapt the substitute component. After the first project, however, has switched already to that given substitute component, we can learn from this other project the steps that have been taken in order to adapt the substitute component and reuse this kind of adaptation code. In order to achieve this functionality, a technique such as provided by type-based adaptation [6] can be used.

3.1 Component Rank

There are two kinds of technology elements needed to find alternative components.

1. Find a set S_C of components which have similar functionality to the replaced component C in D.

2. Find the most reliable component in S_C.

To resolve these issues, we have developed a model called component rank [8]. In this model, the search space for the components is represented as a directed graph. Each node in the graph represents a component. Each edge shows a use relation from a component to another in the sense of a method invocation, instance variable access, and so on. The component rank for a component is then the sorted order of the component by its eigenvalue of the adjacent matrix of the graph. This ranking intuitively shows significance and reliability of the components in the search space, i.e., a component with many incoming use edges from higher ranked components has a higher rank. A component used by many other components inside the project or other projects have many incoming edges, and it will generally have a high rank.

There are many cases that a single component developed in a project is repeatedly duplicated and reused in later projects with slight modifications. In order to identify the duplication, we define syntactical similarity among components. Various mechanisms have been used to identify syntactical similarity among the source codes of components, such as code-clone detection, distance computation by *diff*, and various metrics-value computation (e.g., LOC or complexity). Very similar components are merged into a single node in the graph so that the effect of multiple duplication will be removed.

A component search system has been developed using this model. In this system, a Java class is a component, and the system has many features, such as keyword search, use relation trace among classes, various software metrics computation, code-clone detection, and so on.

To find the alternative components using the component rank system, the following process will be applied.

1. Find S_C in the search space.

   (a) Locate C in the search space. There are two possible ways to do this. We may browse the package hierarchy for the target C, or give keywords which will uniquely extract C.

   (b) Search all similar components S_C. The system has already collected and merged all syntactically similar components into a single node in the graph. So the collected components for C are the member of S_C. Also, functionally similar components are collected by the keyword search mechanism of the component search system. Unique names in library-call statements or in comments will be used as the keyword.

2. Compute the component ranks of all components in S_C, and pick up a high-valued component as reliable one. The system lists up the components in the sorted order of the component ranks, and the developer checks each component from the top until a satisfactory component is found.

3.2 Component Substitution

In the previous section, we have seen how a substitute component can be located on the basis of the knowledge of other projects. Once a substitute component has been located that provides the same functionality as the original component, it needs to be adapted. This is because it is unlikely that another implementation C_{subst} of the component C provides the same interface as the original component C_{orig}.

One option for the developer is to modify the implementation of C_{subst} to make it fit his needs. This approach, however, defeats the purpose of component-based development as it no longer allows the component C_{subst} to be maintained separately. Otherwise, the implementation of C_{subst} would have to modified whenever a new version made available. In order to avoid this problem, developers typically implement small wrappers that adapt the component in a way such that it provides the interface required by their application (i.e., that of C_{orig}).

Once, a project has already located a substitute component and the developers have already implemented the necessary wrappers in order to provide the interface of the original component, it would be more efficient if other projects could simply reuse these wrappers. This can be achieved by putting them into a shared repository where they can be queried for as information about the component interfaces they wrap, respectively provide. By doing so, this repository can be queried by other projects. Hence, adapters become first order objects and can be reused in a way similar to component implementations.

This kind of infrastructure is provided by type-based adaptation which we have presented in [6]. In addition to storing wrappers in a repository, type-based adaptation can only determine when adapters can be combined in order to provide more powerful adaptations. In fact, adapters can be combined when the interface provided by one adapter is the subtype of the interface required by another adapter. In certain situations, this relationship may be relaxed as we have shown in [5].

Type-based adaptation only requires the ability to identify the interfaces required by a project C_{orig} and that provided by a component C_{subst}. Both can be identified on the basis of project data available in the CVS repository as well as the project's inter-dependencies. By using this information and the adapter's stored in the repository, type-based adaptation can automate the adaptation process by deciding when a adapters are needed and how they are to be applied.

More importantly, type-based adaptation can determine when it is necessary to chain several existing wrappers to effect an adaptation that is more powerful than any one existing wrapper can do by itself. This ability to chain wrappers together greatly increases the power of the process and requires many fewer wrappers to be written by the programmers. As we have mentioned before, we only have to define rules on when two wrappers may be combined. In the simplest case, this is the case one wrapper provides the interface that can be used by another wrapper and hence, they may be combined.

4. Related Work

Mockus, Fielding and Herbsleb present a case-study about Open Source Software projects in [12]. They used email archives of source code change history and Bugzilla problem reports to analyze the overall community and development process such as the code contribution, problem reporting, code ownership, and code quality including defect density in the final programs and problem resolution capacity. However, they only used this data to compare the open source development methods by looking at the Apache and Mozilla projects with the traditional commercial one. Our approach, however, focuses on the knowledge surrounding multiple projects and to try to learn from these projects.

Another approach that takes modification and problem reports into account is presented by Fischer, Pinzger and Gall in [3]. They use this data to analyze the evolution of a given software project and to track and to identify hidden relations between different features of the software system. Unlike our approach, however, they only take a single software project into account and do not try to identify relationships between multiple different software projects.

Component rank was first inspired by the famous document search engine Google [4, 13]. Google collects various documents from the Internet, analyzes their link structures among the documents, and computes the significance of each document using similar algorithm as ours. For queries of keywords from the users, Google returns the lists of documents containing the keywords in decrease order of their significance.

Google mainly targets general documents such as HTML and PDF, but it also contains source code of software. So it could be used as the software component search. However Google generally returns less precise lists than the lists made by the component rank. This is because Google does not have any mechanism for the source code analysis such as use-relation analysis and similarity check done in our component rank computation.

5. Summary

In this paper, we have presented a novel idea that analyzes the modification and bug report data of multiple different projects. By analyzing this data, it is possible to deduct knowledge used by the different projects that if shared among the project can lead to improved software engineering practices. A benefit that we have shown in this paper, is that projects can benefit from other projects that make use of the same or similar components.

By identifying projects that use the same component implementation C_i of a component C, it is possible to share knowledge surrounding the component across multiple projects. For instance, if one of the projects using C_i uses an updated version of this component or replaces it with another component providing the same kind of functionality, this knowledge can be shared across projects. Hence, it allows a component C_i to be replaced with a superior component C_j more quickly within a large number of projects. Such kind of knowledge is especially of importance when the component implementation C_i no longer fulfills the requirements of the project. In this case, it is necessary to locate a substitute component c' that is able to meet the projects demands. As we have seen, using our component rank system it is possible to determine the usability and flexibility of different software components and based on that knowledge to infer which component implementation C_j should be used to replace the original component C.

Another challenge that is typically encountered is that a substitute component may not provide exactly the same component as the original component used for a given project. In such a case it is necessary for a given component C_j to be adapted in order to meet the requirements of a given project (i.e., to provide the same interface as the original component C_i). Such adaptation, however, can be easily accomplished using type-based adaptation. As we have explained, type-based adaptation allows the reuse of code that has been implemented in order to adapt a substitute component C_j in order to meet the requirements of the originally used component C_i.

References

[1] J. Dinkelacker and P. Garg. Corporate Source: Applying open source concepts to a corporate environment. In *1st ICSE International Workshop on Open Source Software Engineering*, 2001.

[2] J. Dinkelacker, P. Garg, D. Nelson, and R. Miller. Progressive Open Source. In *Proceedings of the International Conference on Software Engineering*, Orlando, Florida, 2002.

[3] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 90–99. IEEE, Nov. 2003.

[4] The Google website. `http://www.google.com`.

[5] T. Gschwind. Automated adaptation of component interfaces with type based adaptation. Technical Report TUV-1841-2003-14, Technische Universität Wien, Apr. 2003.

[6] T. Gschwind. Type Based Adaptation: An adaptation approach for dynamic distributed systems. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*, pages 130–143. Springer-Verlag, 2003.

[7] T. J. Halloran, W. L. Scherlis, and J. R. Erenkrantz. Beyond Code: Content management adn the open source development portal. In *3rd ICSE International Workshop on Open Source Software Engineering*, 2003.

[8] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component Rank: Relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, 2003.

[9] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic Categorization for Evolvable Software Archive. In *International Workshop on Principles of Software Evolution*, pages 195–200, In conjunction with ESEC/FSE 2003, Helsinki, Finland, 2003.

[10] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic Categorization Tool for Open Software Repositories. In *Workshop on Open-Source in an Industrial Context*, In conjunction with OOPSLA 2003, Anaheim, CA, 2003.

[11] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. On Automatic Categorization of Open Source Software. In *3rd Workshop on Open Source Software Engineering*, pages 79–83, In conjunction with ICSE 2003, Portland, OR, 2003.

[12] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford Digital Library Technologies, Jan. 1998. `http://dbpubs.stanford.edu/pub/1999-66`.

[14] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 1999.

[15] SourceForge.net. `http://sourceforge.net`.