IEICE TRANS. INF. & SYST., VOL.E95–D, NO.5 MAY 2012

SURVEY PAPER

A Survey on Mining Software Repositories

Woosung JUNG[†*], ***Student Member***, **Eunjoo LEE**[††a)], ***and*** **Chisu WU**[†††], ***Nonmembers***

SUMMARY This paper presents fundamental concepts, overall process and recent research issues of Mining Software Repositories. The data sources such as source control systems, bug tracking systems or archived communications, data types and techniques used for general MSR problems are also presented. Finally, evaluation approaches, opportunities and challenge issues are given.

key words: *mining, software, repository, extraction, change, evolution, analysis*

1. Introduction

The size of the data related to software projects is increasing, and thus overwhelming developers and maintainers. Recently, researchers have started to mine software repositories to get a better comprehension of their continuously changing artifacts that are related to long term projects. Thomas Zimmermann said that "Learning from past successes and failures helps us create better software", which best describes one of the ultimate goals of the MSR (Mining Software Repositories) [1]. However, learning from history is not a simple process because software evolves and there are various kinds of data sources.

Most of the cost of the software projects comes from reusing components or maintaining legacy software systems, and not from new developments. Thus, knowledge or patterns from the project history are very useful for software evolution. General activities such as adding or modifying user requirements, changing system environments and correcting software for bug-fixes keep software evolving. However, these works on software evolutions are time-consuming and error-prone even though they could be supported by the previous patterns in the project history.

The MSR field becomes critical to support maintenance, improve the quality of software process and empirically validate various research ideas or techniques. The major goals of the MSR are manifold:

- Supporting software maintenance
- Software process improvement
- Empirical validation of new ideas in software engineering fields
- Predicting defects or detecting inconsistencies

These goals can be accomplished by achieving deep insights about software development and software evolution with the help of the MSR. The goals are very closely related to the analysis methods of the MSR. More explanations about the issues are represented in Sect. 5.

Meanwhile, the first international workshop on MSR was held at the International Conference on Software Engineering (ICSE) in 2004. After four years, MSR became a Working Conference in 2008. The research issues vary from predicting bug patterns to visualizing software evolution. Most MSR analysis techniques for MSR are based on machine learning algorithms and statistics. However, software engineering knowledge is also required to deal with the data or analysis such as code patterns or dependency analysis.

In this paper, we investigated the existing MSR literatures in view of the MSR process. Most of the MSR data are not just from one snapshot of a source code but are from a series or set of codes and documents that have complex relations to each other. As MSR starts with the extraction of the concerned data from various large repositories such as source control systems, bug tracking systems or archives of communications, and so on. The starting point of the MSR becomes to analyze and understand the data sources to extract MSR data. After extracting, the data is transformed into various formats such as text, tree, graph, and vector. Appropriate mining algorithm is selected to process the transformed data and to execute their tasks. In this work, the published literatures before June 2011 have been surveyed. The concrete review questions are as follows:

- **Data extraction**: From where was the raw data extracted?
- **Processing**: What type of data were handled in the MSR process?
- **Analysis**: How are the data analyzed? (algorithms and concrete tasks)
- **Evaluation**: How are the MSR results evaluated?

That is, we divided the MSR process into data extraction, processing the data, and analyzing with the mining algorithms and explained several issues for each phase. We also categorized existing studies according to the types of

Manuscript received July 8, 2011.
Manuscript revised December 16, 2011.

[†]The author was with Software Capability Development Center, LG Electronics, Seoul, 137–130 Korea.
[††]The author is with Kyungpook National University, Buk-gu, Daegu, 702–701, South Korea. (Corresponding author)
[†††]The author is with Seoul National University, Gwanak-gu, Seoul, 151–742, South Korea.
[*]Presently, with Chungbuk National University, Heungdeok-gu, Cheongju Chungbuk, 361–763, South Korea.
a) E-mail: ejlee@knu.ac.kr

DOI: 10.1587/transinf.E95.D.1384

their tasks and presented concrete tasks and evaluation techniques.

To our knowledge, there are a few survey papers of MSR, though several literatures exist which describe a part of the various MSR issues. Kagdi et al.'s work [121] is the most referred paper for MSR survey. Kagdi et al. provides an overall survey and substantial taxonomy of MSR, with four dimension of the type of repository (what), purpose (why), the methodology (how), and evaluation (quality) [121]. The taxonomy is expressive and the survey results are well structured. In this paper, the process of MSR is a basis of description, which helps readers understand the MSR issues according to the MSR process, while 'what, why, how, and quality' were the perspectives of [121]. In this work, the information source is described more detailed and the recent trends are reflected. The type of processing data and various mining algorithms have been well classified, which is not the point of [121]. As the surveyed literature in [121] has been published before August 2006, the necessity of new survey paper increases that reflects the recent trend in MSR, due to the growth in the MSR area.

This paper is organized as follows. Section 2 shows the basic concepts and the overall process of the MSR. The processes and the related issues for extraction, processing, analysis and evaluation are represented in Sect. 3 through 6. Opportunities and challenges in MSR are presented in Sect. 7. Finally, the conclusions are drawn in Sect. 8.

2. Overview

2.1 Background and Scope

The Mining Software Repositories is described as "a field which analyzes the rich data available in software repositories to uncover interesting and actionable information about software systems and projects [5]". The definition of MSR is similar to that of data mining, which is defined as "the process of automatically discovering useful information in large data repositories [6]". Actually, data mining is a more general field than MSR. Most analysis of data mining is based on numeric, nominal or text data which are related to business concepts. However, the information of software engineering area is not limited to such types. MSR requires software domain knowledge for the analysis because its sources mostly come from code files, bug reports, design documents or other special kinds of development archives. Extracting and processing these data are not easy without software engineering domain knowledge and cannot be understood just with statistics. There are lots of specialized techniques or tools for parsing, extracting software data. For example, *ANTLR* [50], *JDT* [51] is used to parse Java source codes. *UMLDiff* [78] provides change facts of object-oriented design model between the two releases, and *SoftChange* [92] extracts software trails like version releases, mailing lists, and version control logs. *Rationalizer* [128] extracts history data and visualizes them in various views. Those extractors have been introduced in Sect. 5.2. MSR is more than just a kind of data mining whose sources come from software.

The definition of reverse engineering in ISO/IEC 24765:2009 is "a software engineering approach that derives a system's design or requirements from its code [7]". In the sense of analyzing and extracting meaningful structures or patterns, the MSR approaches are similar to those of reverse engineering. Reverse engineering supports developers in finding defects or comprehending complex systems by generating or recovering models and architectures. Usually, a snapshot of source code is analyzed and abstracted in reverse engineering. However, MSR considers series of data changes from the history of projects, not just a single snapshot. Additionally, the sources of MSR are more various than reverse engineering because not only the code files but also developers' social networks, design documents, bug reports are used for the analysis. Therefore, the MSR data is generally much larger and complex than that of reverse engineering.

We mainly surveyed the literature presented at the representative workshop in MSR, IEEE/ACM International Workshop on the Mining Software Repositories, from 2004 to 2011. Especially, we studied in detail the papers from 2007 to 2011 which were not surveyed in [121]. Besides them, this paper incorporates the works presented in the main venue of software engineering, such as, ACM/IEEE International Conference on Software Engineering (ICSE) and Automated Software Engineering (ASE), IEEE International Conference on Software Maintenance (ICSM), ACM International Symposium on the Foundations of Software Engineering (FSE), and some papers related to MSR topic which have been published in several SE journals like IEEE Transaction of Software Engineering, Journal of Software Maintenance and Evolution: Research and Practice, and so on. As we intend to investigate influential literatures about software evolution, the scope of this work includes literatures which have been conducted on software systems that have multiple snapshots, like [121]. However, a few studies have been included though the target systems in them have a single snapshot. For example, using specific dataset [68], validating existing tool for bug detection [108], a tool to support developers which record editing operations [106], are the cases. They did not process historical data of target systems, but they also supported MSR activities and multiple snapshots were not required to them.

2.2 MSR Process

The general process of the MSR is composed of the following steps, as is shown in the Fig. 1. The process is very similar to data mining. The objects and processes are represented in angulated and rounded rectangles, respectively. MSR researcher can be an actor in Fig. 1.

Table 1 shows the outline of issues which are described in this work. *CVS* and *SVN* are based on centralized system and *Git* is distributed system where offline work is enabled and the execution speed is fast. MSR approaches of meta-repository such as, FLOSSMole [180], [182] and FLOSS-

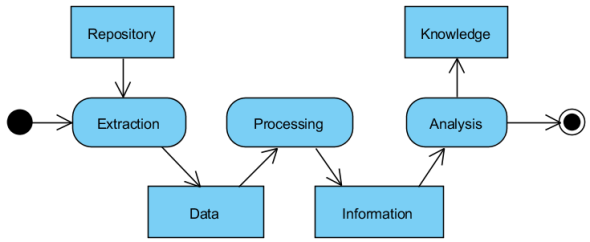

Fig. 1 The general process of mining software repositories.

Table 1 Issues in the general MSR process in Fig. 1.

phase	input(source) output(target)	issue
Extraction	repository raw data	Type of repository -**version control system**: CVS, SVN, Git (CVS, SVN: centralized, Git: distributed) -**bug tracking system**: BugZilla, Jira, Trac -**archives of communications**: emails, mailing lists, messenger, forum -**integrated environment**: Mylyn, IBM Jazz -**code repository**: Google code, SourceForge -**other sources**: design documents, deployment log, crash report
Processing	raw data processed data	Types of handling data -**source code**: text, tree -**natural language**: comment, bug reports, mailing list, etc. -**graph**: source code, relations among developers, etc. -**vector**: attributes representation
Analysis	processed data helpful knowledge	**Data mining algorithms** - classification, regression, association, clustering **Concrete tasks** - bug, change, team-activity, validation, source code com-prehension, understanding development or evolution

Metrics [179], [181], which try to extract various data from many different VCS (version control system), the differences between the repositories should be considered. However, in view of the general MSR researchers, the physical difference is less important. In other words, they usually focus on high level functionalities like obtaining source codes in specific revision, update, and commit. A variety of communication data including chat log, messenger, emails, and forums, can give meaningful information about projects. *IBM Jazz* and *Mylyn* recently arose in the MSR study. Strictly speaking, *Mylyn* is not an integrated environment but a plug-in of Eclipse, however, we classified it into 'integrated environment' because we regarded it as a component of the environment. Additionally, design artifacts and runtime artifacts including deployment log and crash reports can be used as data sources. For more detailed explanation, refer to Sect. 3.

Software-related data such as source codes, bug reports, communication messages, editing events or work items can be extracted from those repositories. Some of them provide commands, APIs or tools for the extraction. After the extraction, they are properly processed to effectively find patterns or rules. For example, some of the text-based data requires tokenization, removal of stop-words and stemming before they are used for analysis. Sometimes, source codes need to be abstracted with heuristics because of the cost caused by the high complexity of extracted data such as abstract syntax trees. Once they are processed and optimized for analysis, various techniques such as association, clustering or visualization can be conducted for obtaining patterns, rules or knowledge. These results are used to support developers or maintainers in planning future projects.

3. Extraction

MSR begins with data extraction, and the extracted data can be classified based on the types of repositories. Data could be collected from one or more various data sources such as source control system, bug tracking system, design documents or archives of communications. However, the majority of research focuses on source codes or bug reports that can be extracted from version control systems and bug tracking systems, respectively. About 80% of the published works in the proceedings of MSR from 2004 to 2011 focus on the source code and bug related repositories. For more detailed information, refer [184]. Other examples of data sources used for MSR are design document, stack traces, mailing list, messages, *IBM Jazz*, *Mylyn*, byte code, project description notes and so on.

3.1 Source Control System

Managing versions of source code is becoming more and more important as the size of project increases. Additionally, most projects are not done by one developer but a team or group of people. Thus, tracking the changes of source code or authors and resolving conflicts in software evolution are necessary for achieving successful collaboration. Source control systems provide such features to developers. Examples of major source control systems include *CVS* (Concurrent Version System) [9], *SVN* (Subversion) [10] and *Git* [11]. From the view point of MSR, code files and histories are obtained from those systems.

CVS and *SVN* are centralized version control systems which use the client-server repository model; however, *Git* is a distributed version control system that uses a distributed model like Mercurial [12], Bazaar [13] or Darcs [14]. Thus, internal structures or methods of storing and managing source codes are different. Table 2 shows a brief comparison of these source control systems. *Git* stores snapshot of each changed file based on diff, without creating new version. The execution speed of *Git* is fast because the products in the servers are replicated and used in the local sites, which enables offline work [138]. Furthermore, as *Git* manages files with three states, such as, committed, modified, and

Table 2 A brief comparison of *CVS*, *SVN*, and *Git*.

SCS	model	offline work	speed	commit unit	revision id
CVS	client-server	No	Slow	Changeset	number
SVN	client-server	No	Middle	Changeset	number
Git	distributed	Yes	Fast	snapshot	SHA-1 hash

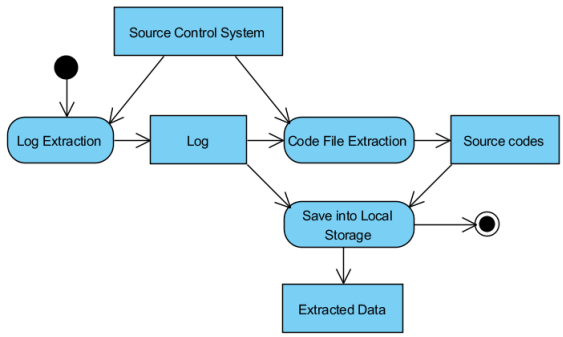

Fig. 2 The general process of data extraction from source control systems.

staged, working directory, staging area, and *Git* repository exists separately. However, general MSR researchers usually focus on high level functionalities like obtaining source codes in specific revision, update, and commit. They share reasonably similar commands for source code management such as *commit*, *add*, *checkout* (or *clone*), *update* (or *pull*). The commands in the parentheses indicate they are *Git* commands. That is, *checkout*, *update*, and *svnsync* in *SVN* correspond to *clone*, *pull*, *clone* in *Git*, respectively.

Figure 2 shows a general approach to extracting data. In order to extract data from source control systems, the log file should be first obtained by using provided commands or APIs of the system. However, creating a local clone repository is recommended before extracting logs for performance reasons. *svnsync* (or *clone*) command can be used for creating a local clone repository. Detail data such as list of changed files, author, date and comments can be obtained for each commit in the log history. By using those data, commands for requesting related source codes can be built and executed on the local repository, which returns a set of changed files. These data are finally inserted into a local database which already has related tables to store the records.

A log from source control system generally includes the following data.

- **Unique id**: Commit order or branch structure can be represented by using these unique identifiers. *CVS* and *SVN* use revision numbers, but they do not provide structural information about branches. However, *Git* effectively represents tree structures or parent commits with its hash values.
- **Date**: *CVS* and *SVN* have only the committing date. However, *Git* provides not only committing date but also authoring date.
- **Author/Committer**: *CVS* and *SVN* provide only committer information. However, *Git* provides not only committers but also authors. Analysis of developers requires identifying authors from logs. Emails and names can be combined and practically used as a key to identify users [137].
- **Comment**: Each commit usually has a comment. Some projects have a rule for comments by using special keywords such as “refactor” or “bug fix”. Sometimes, bug id numbers are included in the comment in order to create relations with the bug reports.
- **List of changed files**: One or more than one file could be changed for each commit. Based on these data, the file types or sizes are also easily obtained. A unique file id should be the composition of commit id and path id because files which have the same path do not always have the same identities due to the different modified date and time.

There are lots of related works using source control systems such as bug prediction, impact analysis, visualizing change traces and detecting clones, refactoring cases or design patterns. This kind of research can support the software process by providing developers insights into the software evolution.

3.2 Bug Tracking System

Today, the size and complexity of software projects are increasing. Thus, a lot of reported bugs should be managed systemically. Bug-related information such as priority, severity, location, how to reproduce bugs, who found the bugs or the status of bugs are stored in bug tracking systems such as *Bugzilla* [19], *Trac* [20] or *ZIRA* [21]. Basically, a bug tracking system manages bug reports which contain detailed descriptions of software failures. However, the structure of the reports is mostly not formal and it is difficult to extract semantics from the original text based reports. Thus, the expected data schema needs to be confirmed before the extraction.

Most bug tracking systems provide web interfaces for managing bug reports and use database management systems such as *MySQL* [22], *PostgreSQL* [23] or *Oracle* [24]. Therefore, necessary records could be extracted directly and stored in separated local databases by accessing the tables of the original databases as long as they are available. A similar approach could be applied in the case that csv or xml files are provided by the bug tracking systems. However, they are mostly unavailable and the data should be crawled and parsed through the web interfaces. Even if they are extracted, some of them are difficult to be identified except some trivial fields such as status, priority or severity. Attached files also should be downloaded and given identifiers to have relations with bug reports. Figure 3 shows the general process for extracting data from bug tracking systems. During the categorization step in Fig. 3, data is classified into two cases: the possible case to expert data in formatted text or database files, or not. When files or data cannot be obtained, crawler or parser is required to get data.

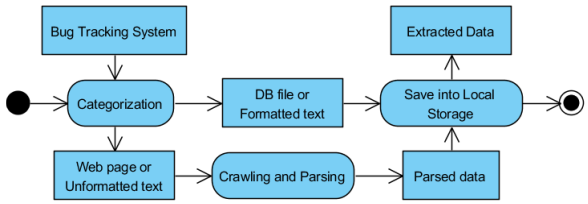

Fig. 3 The general process of data extraction from bug tracking systems.

General data that can be extracted from bug tracking systems are listed below.

- **Bug id**: Unique identifier of each bug
- **Dependency**: Information which can show relations between bug reports or developers such as "assigned to", "duplicated", "other bugs depending on the bug"
- **Version**: The version of software which has the bug
- **Environment**: Operating system or hardware specification
- **Status**: The information which show the status of bug's lifecycle. Generally, it has one status of "unconfirmed", "new", "assigned", "resolved", "verified", "closed", and "reopened".
- **Resolution**: The result of handling the bug report. Generally, it has one status of "fixed", "invalid", "won't fix", "later", "remind", "duplicate", and "works for me".
- **Priority**: Generally, it has one of five levels from "very low" to "very high".
- **Severity**: The impact of the bug. The highest severity is called "blocker" where you cannot even test a problem. "critical" is the next highest case that can break a program and cause data loss. The lowest severity is "trivial" which is related to requirements enhancement.
- **Patches**: Information for fixing the bugs. Most of them are attached files that have uniform diff formats, which can be applied for automatic fixes. Sometimes, descriptions of how to fix the source codes are included.
- **Stack traces**: Execution information which is considered as one of the most important factors for debugging by developers. It can give direct hints for finding causes of defects in the case of software crash. It includes not only exceptions or error messages but also other detail execution information such as call dependencies. Each data can be parsed and extracted by using regular expressions.
- **Source code**: Example codes which are helpful to find the problem or to fix the bug.
- **Descriptions**: Causes or symptoms about the bug, reproducing procedures or solutions to the problem; usually written in natural languages.

According to a survey to find out features of a good bug report [25], most developers considered that reproducing steps, stack traces and test cases are most helpful for debugging. However, users thought they were difficult to provide. These results could be used not only to support developers for designing better bug tracking systems but also to automatically distinguish the good bug reports from bad ones.

In general, the extraction cost or complexity of a bug tracking system is higher than those of a source control system because the structure of a bug report is difficult to be predicted, compared to the change log or source code. Bettenburg et al. effectively generated a common structure of bug reports by using a tool named *InfoZilla* [26] in order to support the mining process of bug reports. They also figured out that the quality of the bug report is improved by merging the duplicate bug reports, not just by eliminating them [27].

For conducting MSR tasks such as change analysis, defect prediction and setting expertise for bug reports, it is important to link bug reports and source repository [132]. Gyimothy et al. presented a technique to link bugs from *BugZilla* database to source code classes [130]. At first, they filtered overall bug database to remove unnecessary data, and then, they allocated the bugs to an area in the codes by analyzing the patch files. As the patch files contain several information like, changed file name, the number of removed lines, and so on, it is possible to determine the change interval in each file. Bugs are matched onto the releases from the date of bug reporting to bug modifying. For each bug, they searched the class in the specified releases, of which interval the bug overlapped. Finally, on-the-fly classes were removed because they had no bugs. SZZ algorithm [58] has also been commonly used. SZZ algorithm is composed of syntactic and semantic level. In syntactic analysis, they inferred links between transactions and bug reports. To do this, syntactic confidence, *syn*(0-2), is assigned to log message by token analysis. High *syn* indicates that the log message is highly possible to be buggy. After that, the link is validated using bug report data, which is the semantic analysis. The semantic confidence, *sem*(0-4) is assigned to the links. *syn* and *sem* is computed in the heuristic way. For example, *syn* increases by one when there are predefined keywords like *bug*, and *sem* increases by one when the author in transaction is allocated to the corresponding bug. Kim et al. proposed the extended SZZ algorithm [131]. They pointed out that the built-in annotated feature of the SCM (software configuration management) on which SZZ depends is insufficient, and the modifications do not always get fixed. Kim et al. built annotation graph where nodes and edges denote code lines and evolving from nodes, respectively, to supplement the insufficiency of annotation in SZZ. And then, they excluded the changes in format or comment, and addition or removal of blanks to reduce false positives. Furthermore, they eliminate an outlier, excessively modified file at a revision, because the modifications are less likely to fix bugs.

3.3 Other Sources

Even though most MSR research focuses on source control systems or bug tracking systems, the mining sources are not limited only to them. Archives of communications, *UML*

diagrams are also interesting data sources for MSR. Especially, *IBM Jazz*, which provides collaborative environments for a whole lifecycle of software development, manages not only team organizations but also work items including history of changes and event logs. On the other hand, *Mylyn* can provide developer low-level interactions such as selecting menus or editing methods.

Archives of Communications

A lot of data related to communications are generated via email, messenger or off-line meeting, which are very useful for identifying the structure or efficiency of teams. Archives of communications not only affect the quality of software [34], but also can support the prediction of failure [35].

Communication data based on text is important though speech is the most principal communication data, due to the availability of text data. In the case of email, the sender, receiver, subject, content, date, time, priority, and attached files are available for data extraction. These can be directly obtained from the mail box files if they are accessible. Otherwise, crawlers should be applied for web mail clients. Thus, the general process for extracting data is similar to that of the bug tracking system. However, the approaches for archived communications are more dependent on text mining based on natural language processing.

Network graph generated from the communication data often supports "Conway's law [36]". Thus, local interaction history is often analyzed in order to improve the quality of software process based on the structure of developers' organization [37]. Especially, the structure of developers is relatively dynamic in the case of open source project because the participations are free [149]. The mailing lists in open source projects play an important role for communication [145], and they provide helpful information for developers or projects [139]. Yu et al. analyzed the associated social networks of developers based on their interactions extracted from two open source projects, *Linux* and *KDE* [38]. They considered the channel directions of messages or threads, and assumed that one-way is a service relationship and two-way is a collaboration or coordination. They also defined evolution models and predicted the dynamics of social networks by using bandwidths and interaction degrees based on the size of messages.

Recently, *IRC* meeting is increasing in software development projects. Thus, analyzing the messages in the conversations of developers is a new challenge. The first study of MSR on the open source project was conducted by Shihab et al. by extracting the message volumes, the size of participants and their activities from *GNOME GTK+* project [39], [40].

Design documents

MSR data are not limited to source code or text data. Software artifacts such as *UML* diagrams, which include abstract models of packages, classes, components, sequences or activities, are also interesting data sources for MSR analysis. In order to extract information from the *UML* diagrams, using *XMI* [41] is one of the easiest ways because most major *UML* modeling tools like *Visual-paradigm* [42], *Enterprise Architect* [43], *IBM-tau* [44] can export the diagrams as *XMI* format. The extracted data of elements and relations from the exported files can be stored into the database tables that have been defined based on the XMI schema.

Based on the extracted data from *UML* diagrams, various analyses such as prediction or association are possible. Nugroho showed that the quality of Java class can be effectively predicted based on metrics such as the detail levels of messages and import coupling, which are obtained from sequence diagrams and class diagrams, respectively [45].

IBM Jazz

MSR research aims at extracting data and knowledge from separate source codes, code changes, bug reports, emails or communication messages. However, their relations are missing and it is difficult for researchers to organize or combine their separate data sources. *IBM Jazz* [3] is a collaborative software engineering environment that provides full traceability among all the artifacts of software development. Thus, important mining sources such as codes, bug reports, work assignments, changes and tests are formally related to each other. This feature enables researchers to analyze data or predict defects with clear relationships without using the mapping heuristics among data sources. Recently, fundamental studies to extract data from Jazz repositories have been conducted [46], [47].

There are four major extraction approaches for *Jazz* repositories and they have trade-offs among their strategies. Direct access to the database of *Jazz* provides the most authority for the repositories. However, it also requires the highest cost for understanding the complex schema of the database, which may result in generating errors or faults while handling the repositories. Contrary to this, extracting data from only automatically generated reports provides the safest way with low cost. It could be intuitively understood, but the extracted data will be restrictively available. Client API or Server API could be used to extract data from *Jazz* more safely with lower cost than directly accessing the database but more effectively than using reports. Table 3 shows a summary of the extraction methods of *IBM Jazz* repository. In directly accessing database, the cost is high due to the need of understanding database schema, including the types and meanings of table and field, and their relations. Reports require little cost, because they provide final data whose meanings are clear. In case of Server API or Client API which is located between the database and the reports

Table 3 Extraction methods of *IBM Jazz* repository.

Method	Cost	Accessibility	Safety
Database	Very High	Very High	Low
Server API	High	High	Middle
Client API	High	Middle	High
Reports	Low	Low	Very High

in the layer structure, the cost is lower than the database but higher than the reports, as it is not required to understand the schema. As it is possible to directly access the table and field in using database, the accessibility of the database is very high. With the opposite reason of database case, the accessibility of the reports is very low. It is possible to violate integrity of database and to extract wrong data or missing data on account of misunderstanding schema, in directly accessing database. However, using the reports reduce this risk because they are normally extracted via internal logic of Jazz. For server API and client API, accessibility and safety are affected by the order in the layer where the bottom layer is database, the top is reports, and server API and client API are located between them in order. For example, the accessibility of server API is a little higher than that of client API.

Mylyn

Mylyn [4], an Eclipse plug-in, collects information about programmer's activities such as editing files, methods and selecting menus with time stamps or identifiers for each event. It can manage eight kinds of interaction events such as selection, edit, command, preference, prediction, propagation, manipulation and attention. Method level interactions are also available for the data extraction. Thus, the data extracted from *Mylyn* are very useful for analyzing the relations between the tasks of developers and the resources, which include not only classes or files but also methods. It also provides degree-of-interest (*DOI*) values representing how frequently and recently the elements in the tasks have been accessed. Thus, the elements with negative *DOI* value can be ignored for the analysis [48], unless the target of analysis is to find the elements which have not been frequently used. *Mylar* [49], the origin of task context for the Eclipse development environment, has changed its name to *Mylyn* since 2007.

Miscellaneous

Most of the MSR studies are focused on source control systems and bug tracking systems. However, the data sources for MSR are not limited to them, neither are they exclusive to each other. Thus, other data sources such as program execution information [30], crash reports [133], [185], and test cases [134] can also be introduced for MSR research. Execution information enables to reflect abnormal behavior which had not been detected by the bug report and it is little influenced by the variation of natural language [30]. Crash reports can be used for bug fixing or crash triage, because they have stack traces and run time information about when the crash happened [133], [185]. Deployment logs contain execution information of one or multiple sites and they are increasingly used in MSR [136]. Furthermore, code repository site such as *Sourceforge.net* [186] or *Google code* [187] is a data source where massive software projects are provided [136]. *SourceForge* is a code repository which is web based, and it hosts many projects which are in high level compared to *CVS*, *SVN*, and *Git*. *SourceForge* naturally utilizes several version control systems like *CVS*, *SVN*, *Git*, and so on, to control multiple versions. Thus, it may be helpful for analyzing multiple open source software to use code repository. *Google code* is useful for studying patterns of code, because it is possible to search codes using several conditions like package, language, class, function, and licenses and to identify codes of files for the various projects. These deployment logs or source codes from multiple source code repositories could be analyzed together with other data sources such as bug tracking systems at the same time [135].

4. Processing

4.1 Source Codes

Source codes can be regarded as a set of text strings. However, they have tree structures based on syntax. Thus, an abstract syntax tree is often used when token-level analysis is required. Table 4 shows the differences between text based source code and abstract syntax tree. In processing text, the data type is string. However, the data structure of tree and operations of them are complex because tokens and edges are dealt with in the case of tree processing. Tree enables to analyze data dependency and control dependency using the tokens and edges which denote relations, but it is hard to analyze them in text. For these reasons, accurate analysis, such as size, complexity, and dependency, is possible in tree but they are difficult in text.

Text

The cost of processing text-based source code is much lower than that of tree-based source code. However, dependency analysis or structural matching is not applicable for raw text-based data. Even if they are possible, precision is very low compared to the tree-based data. In spite of their lack of applicability and accuracy, text-based source code is often used for structural matching with a technique of replacing specific substrings with special characters such as "*" or "?" [15]. After abstracting text-based source code, regular expressions are used for structural analysis.

In order to analyze the code change patterns, changed pairs of files should be produced from the history logs. And then, the differences of codes between adjacent revisions can be calculated for each pair via text-diff tools.

In general repository systems, line-based code differences can be generated for each change if proper log options are applied. The results show the locations of added, removed positions of the changed files by attaching "+" or "−" character in front of the changed code lines. However, additional processes are required in order to get added, removed or modified code hunks because the result is composed of

Table 4 A comparison between text and tree data of source code.

Type	Unit	Complexity	Dependency	Precision
Text	line	low	no	low
Tree	token, edge	high	yes	high

text strings which have multiple sets of such changes. The code hunks can be extracted by checking the sequence of "+" or "−". For example, if continuously changed parts have a pattern of "−" or "+", then the "−" part can be considered to be modified to "+" part. Eventually, each change has multiple code hunks.

Tree

Tree-based code requires a much higher cost of handling than the text-based one. However, it enables detailed approaches such as dependency analysis or structural comparison on the token level. Tree-based *diff* is used for detecting detail changes which are mostly based on heuristic algorithms in order to reduce calculation cost [2]. The tree-based *diff* tools are closely related to similarity or distance metrics and could be used for detecting clones. The differences between trees are not just text strings, but the set of changed pairs of tokens and edges. Thus, it is much more complex and difficult to calculate their differences compared to the list of added or removed text lines. Generally, if move operations are considered, calculating an edit distance between two different trees is NP-Hard [2]. Thus, it could require too much time to calculate the differences between every changed file because the data size for the MSR is usually very large. As a result, it creates the scalability problem. Therefore, this kind of detail analysis should be done only in the area of concern, not for the whole source codes.

Source control systems basically do not support tree-diff operations. Thus, text-based source codes should be transformed to abstract syntax trees before being analyzed. *ANTLR* [50], *JDT* [51] can be used for parsing Java source codes or *java2xml* [52] can be used for transforming the source code into an xml file, which has tree representations for the codes. Further, efficient xml diff tools [53], [54] can be applied for calculating the differences between two xml files. However, xml differences are difficult to represent in intuitive formats because most of xml diff tools use paths or pointers to represent the changed sub trees. As a result, calculating the differences directly from two abstract syntax trees and storing them in neat format for long time project history is challenging. Therefore, heuristic approaches such as comparing token counts or abstracted text strings, which are generated from the trees are sometimes more effective. They can be used for code analysis if detail level analysis such as dependency analysis is not required. However, tree-based code is necessary in order to increase precision, especially for structural methods such as finding dominant usage patterns or code examples from the historical data. These patterns or examples can be used to detect suspicious code usage or to guide the developers to make better codes from the learned knowledge.

4.2 Natural Languages

Most of the major data extracted from comments, bug reports or archives of communications are text-based format in natural languages. The types of bugs can be classified with the words extracted from bug reports in a heuristic way [140]. The types of changes can be determined with commit messages [144]. The characteristics of projects are derived from work descriptions [143] or mailing list based on word count [139]. General processing steps for such data include the following.

- **Tokenization**: The original large text strings are divided into a set of tokens. Parsed tokens or simply separated tokens can be obtained.
- **Removal of stop-words**: Meaningless tokens such as "a", "an", "the", "in", "of", "this", "that" and etc. are eliminated, which leaves only meaningful tokens that have semantics.
- **Stemming**: The tokens with the same meaning but different expressions are transformed into a unified token. For example, "looked", "looks", "looking" are changed to "look".
- **Generating a bag of words**: Unordered set of words in each file are transformed to a set of tuples. Each tuples has a token and the token count. Sometimes, the count is replaced with a weight value based on term frequency and inverse document frequency.

The natural language is eventually transformed to dynamic vectors whose attributes are tokens. And then, they are used for further analysis such as classification, prediction or clustering.

4.3 Graphs

Source code elements can have relations such as call, use, dependency and assignment, which can compose a network between tokens [141]. The developers also have relations and a social network could be created among them. These networks are represented in graphs [176]. Table 5 shows some examples of basic relations which compose those graphs. However, there is lack of detail information in the graphs such as modified date, size, and impacted code hunks. For example, *Fix*(*Jack*, *foo1.java*) which means that Jack fixed the foo1.java file, does not tell such specific information. In brief, a graph can only show the relations between entities and further information should be managed somewhere else. That is, it is possible to store additional information like relating table with foreign key in RDBMS. Of course, nodes and edges can apparently store these information with their attributes.

Table 5 Examples of 3-tuple for graphs.

Entity 1 relations entity 2 examples
developer communicate developer Email(Dave, Jack), Message(Jack, Bob)
developer use 'software artifact' Fix(Jack, MyClass.java), Remove(Dave, Yours.xml)
'software artifact' include 'code element' Include(Spec.doc, requirement1)
'code element' dependency 'code element' Call(foo1, foo2), Include(MyClass, foo3), Assign(var1, var2)

Those data for graphs can be generalized to "Relation(Entity1,Entity2)" which means that "Entity1" and "Entity2" have an order. Relational databases can easily implement this kind of fundamental types. However, sometimes further processing is required for reasons related to performance by transforming a sub-graph into a text string or defining new complex types that are optimized for specific domains. For example, you can create class, method, field or parameter tables for easier source code analysis. Separate tables of statistics also could be defined for special purposes such as visualization.

4.4 Vectors

Specific data such as source code, natural language, and graph, usually have attributes, which could be one of numeric, nominal or text data types. They are often calculated from metrics or values among the predefined categories. The patterns of attributes are important for prediction, clustering or association analysis. The number of attribute is also critical because too large size of attributes could reduce precision. Thus, selecting key attributes is necessary not only for achieving better analysis results but also for simpler models. In addition, some attributes that depend on other attributes should be eliminated for better analysis. When there are dependencies between the attributes, it is not needed to use both of it; by enduring the high computational cost due to large vector size. In addition to cost problem, co-related attributes which has not been reduced may be the cause of biased results. For better processing, it is enough to use one representative of co-related attributes.

Some specific machine learning algorithms are not applicable for numeric type attributes. Therefore, those attributes should be transformed into nominal types by defining some limited ranges and grouping similar numeric values into one category. Normalizations of attribute values are also required to compare data which have different ranges of values. For example, (v-min)/(max-min) can be used for linear normalization where v is the attribute value before normalization, min and max is the minimum value and maximum of the attribute, respectively. The changes of software can be represented with vector [142], entropy can be used as a key attribute to characterize the author contributions per file [55] or to calculate the complexity of changes for predicting error-prone codes [56].

4.5 Discussion

"Garbage in, garbage out" is a phrase emphasizing the importance of input data in order to get high-quality output. Actually, data extraction and processing are very important steps for making analysis easier and improving the quality of the result. Most MSR time is spent for data extraction and processing.

Sometimes, further processing may be required in order to be adapted for tools or environments. For example, the final data should be table styles for *Excel* or DBMS, and *arff* format is required for *WEKA*. Thus, proper data types such as nominal, ordinal or numeric should be defined based on the metrics or probabilities calculated from the software engineering domain knowledge.

While processing fundamental data, outliers could be detected in the data. However, they are not always removed because they could give more interesting results for anomaly analysis. Most of data mining algorithms are robust to noise, but some of them are not. Recently, there has been a research to reduce noise from raw analysis data [57].

5. Analysis

5.1 Data Mining Algorithms

Data mining algorithms are often used in the MSR analysis for source codes, bug reports or software artifacts. For example, classifications or regressions in MSR can be considered as supervised learning problems. Classifying priority, severity, security bug reports or good reports, and predicting defects based on the bug/fix memories belong to these problems.

Bayesian network, rule-based *ZeroR*, tree-based *Id3* or J48 are mainly used for the classification problems. However, in order to use *Id3* algorithm with numeric data, proper transformation should be conducted for the numeric attributes because only nominal values can be applied for it. Support Vector Machine (*SVM*) is also known to be more general and achieves high performance of classification. Neural networks are mainly used for regression problems. Regression is very similar to classification but the only difference is that the output has quantitative values, not nominal values. Association is finding related attributes such as the change coupling issue in MSR. Actually, using historical data is a very effective way for this issue because it could find co-change relations even if there are no traditional dependencies such as data or control flows. *Apriori* is a major association algorithm, which is also only applicable for the attributes of nominal data types. Clustering is a typical unsupervised learning problem, and the major methods are hierarchical clustering, *K-means*, *SOM* (self organizing map) and *EM* (expectation maximization). In the case of using *K-means* or *SOM*, the number of clusters should be known and the cost is higher than hierarchical approach. However, they are known to have better quality of results than *EM*. As clustering is an unsupervised problem, historical data is not always necessary. Examples of related problems are clone detection and grouping components.

Table 6 shows major data mining algorithms for MSR and their related issues. It has been referred from Halkidi et al.'s work [146]. These algorithms are effectively applied to vectors that include numeric or nominal types such as statistical data or metrics. For example, the nearest neighbor algorithm has been used to predict the effort of issue reports in [163]. Decision tree has been used to predict developers' contribution in [145]. In [65], SVM has been applied for the bug triage and in [147], association rule mining has

Table 6 Major data mining algorithms and MSR issues.

Category	Classification	Regression	Association	Clustering
Supervised	Yes	Yes	Yes	No
Input Output	don't care discrete	quantitative quantitative	don't care associated attributes	don't care homogeneous clusters
MSR Issues	priority, severity, SBRs/NSBRs for bug reports, etc.	change rate, # of bug, quality, complexity cost, etc.	change coupling, impact analysis, etc.	clone, code pattern, etc.
Algorithms	SVM, Nearest Neighbor, Decision trees	Neural network	Apriori	K-means, hierarchical clustering, SOM, EM

Table 7 Detailed category of purposes.

keyword	existing approaches
bug	bug fix [58]–[61], [155] duplicate bug detection [28]–[30] prediction [15], [18], [31]–[33], [62]–[64] bug resolvers [65], [66] using information retrieval [68]
change	prediction [69]–[71] refactoring [72], [73] API-change [74], [75], [77], [80], [81] change patterns [83]–[88], [90], [160]
team-activity	developer's contribution [55], [91], [93], [94], [154] experties of developers [96], [97], [149] tool support [98], [99], [128], [151] helpful information [100]
comprehension	visualization [101], [102], [156] identifiers [104], [105], [153] recording operations [106]
validation	metrics [45], [107], [157] tool [108] clones [109]–[112], [150], [159] bug [113], [114]
development& evolution	development [118]–[120] evolution [115]–[117], [152], [158]

been applied for the defect data analysis. Hierarchical clustering algorithm has been used to understand the developer's role in [148], and defect priority has been determined based on neural networks [32]. However, other specific algorithms should be implemented when directly applied to the domain oriented types such as source code.

There are more issues such as sequential pattern or outlier discovery. Sequential pattern analysis focuses on finding relations in ordinal data and it is related to automated code completion or change prediction. Outlier discovery is related to detecting anomalies in the source code or development process.

5.2 Purpose of MSR Analysis

Table 7 presents the detailed category of MSR purposes, their detailed task types, and existing approaches. Table 8–13 for each purpose summarize the existing approaches in view of task, data sources, output, and target systems. Due to space restriction, only a part of it is presented. For entire tables, please refer [161].

Table 8 Existing approaches to support bug-related activities (part).

task	[ref]	data source	output	target system
bug-fix analysis	[155]	Git, CVS	development characteristics	Linux Kernel, PostgresSQL
detecting duplicated bugs	[30]	CVS, bug repository	duplicated defect reports	Eclipse, Firefox
revealing useless phase in defect prediction	[63]	CVS, Bugzilla	empirical observations: the influence of concept drift	Eclipse, OpenOffice, Netbeans, Mozilla
predicting the severity of bugs	[31]	BugZilla	severe bugs and non-severe bugs	Mozilla, Eclipse, GNOME
identifying security bug reports	[129]	Cisco's bug tracking system	security bug reports	four Cisco software systems
bug triage	[65]	CVS, BugZilla	expertise to fix the reported bugs	Eclipse, Firefox

To support Bug-related activity

There have been several studies on bug-fix: empirical study on the patterns for bug-fix [58], [155], automatic bug-fix [59], [60], understanding the bug-fix patterns of hardware project [61]. Sliwerski et al. conducted the empirical analysis about fix-inducing change based on *CVS* log and *BugZilla* [58]. For example, they investigated whether or not some change properties such as specific day or specific working group are actually correlated with problems. The experimental results showed that fix-inducing changes mostly happened on Friday and Saturday in case of *Mozilla* and *Eclipse*, respectively. The number of fix-inducing transactions is about three times that of non-fix inducing transactions. Eyolfson et al. studied the co-relationship between the patterns of commits and the bugginess for those commits [155]. They explored the Linux Kernel and PostgresSQL and found several observations: The commits from midnight to 4 A.M were highly possible to be buggy and the commits of everyday committers were less buggy. They also argued that the influence of day-of-week on commits was variable for each project. Williams and Hollingworth suggested a technique to automatically find and fix bugs by mining bug-fix information in source code repository, especially on the bugs of function-return-value check [59], [60]. Sudhakrishnan et al. studied the bug-fix patterns of hardware projects [61]. As most of hardware projects utilize CM (Configuration Management) repositories, they mined bug-fix history on four Verilog projects and manually defined 25 bug-fix patterns.

There was a text-based analysis approach for detecting duplicate bugs [28], [29]. However, Wang et al. increased the recall of duplicate bug detection by combining the methods from natural language and execution information [30]. Their approach showed 67%–93% recall in Firefox repository, which had been 43%–72% for natural language only.

Prediction is the main subject of mining software repository and predicting bugs has been also widely studied. By extracting bug/fix code patterns from the history, similar code patterns in the future are considered to have high chances of introducing similar bugs. Based on the bug/fix

patterns that had been learned from the change history of source codes, *BugMem* [15] tried to predict the project-specific bugs. The approach is different from others such as *JLint* [16], *FindBugs* [17] which use the static analysis of a snapshot of source codes. *BugMem* is more effective for detecting project-specific bugs, but it has limits in finding some trivial bugs such as missing null checks. It also requires enough change history to use the approach. There was a similar research using change couplings. Four kinds of bug localities such as change entity, new entity, temporal, and spatial localities had been checked whenever the source codes were modified. Further, the locations which are only related to bugs and fixes were stored into a cache [18]. If a similar code in the cache is detected, the developers are alerted for possible bugs. The accuracy was 73–95% in the file, 46–72% in the method level. Nagappan et al. suggested an approach to combine complexity metrics and post-release defect history as to construct the prediction model of post-release failures [62]. They also validated four hypotheses that present the correlation between complexity and post-release defects, by five MS products. The hypotheses and the validation results are as follows:

- H1: The complexity metrics correlate with post-release defects (Supported).
- H2: There exist a single set of metrics which predict defects in all projects (Rejected).
- H3: There exist combined metrics to predict the post-release defects within a project (Supported).
- H4: The combined metrics in H3 can predict defects in other projects (Partially confirmed).

In case of H4, they asserted that the predictors from a project are only useful to similar projects, and not every project.

Lamkanfi proposed a technique to predict the severity of bugs via text mining algorithms, which had been manually predicted [31]. The technique has been applied to three open sources, *Mozilla*, *Eclipse*, and *GNOME*. The results showed that a sufficient size training set makes it possible to predict with reasonable accuracy. There have been studies of predicting the priority of bug reports based on neural networks [32]. Similarly, classifying security bug reports from non-security related reports was conducted based on the text mining approach [33]. Ekanayake et. al revealed that useless phase exists in defect prediction using the notion of concept drift, which invalidates a learned prediction model [63]. As history data is a good predictor of future bugs in the stable phase, however, in unstable phase, it is not the case, resulting in reducing the effectiveness of future effort and resource allocation. They built the defect prediction model for *Eclipse*, *OpenOffice*, *Netbeans*, and *Mozilla*, and they then visualized the prediction quality. The results represent that software systems usually have significant concept drifts in history, and particularly the number of authors editing files and the number of defects fixed by the authors contribute the concept drift and degenerate the quality of prediction. Giger et al. showed that fine-grained source code changes (SCC) is better than the existing line modified (LM) for bug prediction [64]. SCC incorporates semantic of changes, which are not provided by LM. They established three hypotheses: First, SCC is strongly correlated with the number of bugs. Second, SCC is more effective than LM to classify the source files into bug-prone and none bug-prone. Third, SCC outperforms to predict the number of bugs compared to LM. These hypotheses are validated through an experiment on the *Eclipse* system.

There are researches on recommending bug resolvers [65], [66]. Anvik et al. recommended the list of potential developers who can resolve BRs by supervised learning [65]. The past reports of *BugZilla* are applied as classifiers, and they are then trained with project-specific heuristics, not with the direct usage of 'assigned-to' fields in BR. Matter et al. suggested an approach to automatically assign BRs to relevant developers using vocabulary [66]. The expertise of each developer has been modeled by comparing the vocabulary of source codes contributed by the developer and the vocabulary of BRs. The evaluation has been conducted based on a comparison between recommended developers and actual developers. The empirical results incorporate 33.6% top-1 precision and 71.0% of top-10 recall based on investigating the *Eclipse* for 8 years.

Information retrieval is applied to bug localization. Rao and Kak compared five text models, VSM (Vector Space Model), LSA (Latent Semantic Analysis Model), UM (Unigram Model), LDA (Latent Dirichlet Allocation Model), and CBDM (Cluster-Based Document Model), to retrieve relevant files from libraries using benchmarked dataset iBugs [67], [68]. MAP (Mean Average Precision) and 'Rank of Retrieved Files' are used as evaluation measures. In conclusion, a simple model, such as VSM or Unigram shows better performance than complex models like LDA, LSA, and CBDM.

Table 9 Existing approaches to support change-related activities (part).

task	[ref]	data source	output	target system
change prediction	[69] [70]	CVS, BugZilla	the set of changeable files	Kcalc, Kpdf, Kspread, Firefox ([69]) Gedit, ArgoUML, Firefox ([70])
relating API changes to refactoring	[74]	version control system	empirical observations: the influence of API changes	Eclipse, Struts, JHotDraw, Log4j, Mortgage
detecting API evolution	[77]	version control system	API changes by Diff-Catchup Diff-Catchup: a tool to recognize API changes	HTMLUnit, JFreeChart
providing API usage adaptation patterns	[80]	version control system	suitable patterns by LibSync LibSync: API usage code adaptation framework	JHotDraw, JFreeChart
identifying FAC	[88]	CVS	FAC (frequently applied changes)	Tomcat

To support Change-related activity

Many studies have been conducted on change prediction area [69]–[71]. Canfora and Cerulo defined impact analysis techniques based on information retrieval, which notify the set of changeable files using the textual descriptions about newly introduced bugs in the bug repository [69]. In that approach, a BR is linked to a commit message, that is, specific bug id is connected to a set of files. The list of changeable relevant files can be created by querying in the textual description in BR. After that, in [70], precision has been enhanced by 10%, due to the granularity has been detailed from file-level to line-level. However, execution time changed from second level to hour level. Robbes et al. presented a benchmark that is able to evaluate the change production technique with fine-grained change data recorded in IDE, and showed the procedure to estimate existing prediction techniques [71].

Refactoring is a typical cause of change. Weiβgerber and Diehl defined a method to identify refactoring in changes [72]. In [72], line-based differences are mapped onto the difference between syntactic entities like class. Several refactorings such as move/rename class and move field/method, are identified from changes like add/delete/modify. And then, they indirectly correlated the number of bugs per refactoring with the number of change entities, the number of bugs per changed entities, and the frequency of refactorings per changed entities, as to ascertain that refactorings induce less bugs than other changes. They concluded that refactorings are less bug-prone in most cases. Ratzinger et al. investigated the relationship between refactoring and defects [73]. They extracted 110 data mining features from versioning and issue tracking system. The extracted features have been classified into refactoring feature and non-refactoring features, which were utilized as input data for the classification algorithm of the defect prediction model. They showed that the features improve the quality of the defect prediction model. They also presented that refactoring and defects are inversely correlated. Finally, they argued that refactorings play an important part in evolutionary changes to decrease the number of bug-fix and defects.

API changes are deeply associated with refactorings. Dig and Johnson studied the API changes between two versions of framework/library (component) and classified changes into breaking change and non-breaking change [74]. Various data are used such as "change logs, release notes, help documentation, developer interviews, and manual examination of source code differences". Their approach has been applied to three open source frameworks, *ECLIPSE* framework, *Struts*, and *JHotDraw*, one open source library, log4j, and one proprietary framework *Mortgate*. Two versions were analyzed for each framework. The results showed that about 89% of the total breaking changes were the effect of refactorings. In other works, the principle of "behavior preserving" was kept in frameworks or library. However, it was broken in client applications. Henkel and Diwan developed *CatchUp*, which captures and replays refactoring to support API evolution [81]. This is a lightweight approach, not using version control or configuration management system. After capturing the API change, *CatchUp* replays the refactoring when it is applied to client components. Taneja et al. found that 80% of API changes were caused by refactorings, and they proposed an approach to automatically detect the refactorings to automatically upgrade the applications [75]. In the first step, refactoring candidates for two versions are extracted using *Refactoring-Crawler* [76] by syntactic analysis. And then, *RefactLib* refines the results using various heuristics and classified them onto seven predefined refactoring types. Xing and Stroulia studied the API evolution problems in reuse-based software development and suggested "API-evolution catch-up methodology" [77]. In the approach [77], API changes are automatically detected in the reused framework and a relevant substitute for "obsolete" API is recommended based on working examples of the framework code base. The methodology consists of three phases: First, *UMLDiff* [78] automatically detects the change facts of the old and new versions in reusable component framework. Second, the heuristic process is executed in API migration problems to answer the questions with which the client application developers are confronted. Third phase, the client application developers obtain a set of replacement and usage example proposals that are formulated and presented. Those results are visualized with *JDEvAn* Viewer [79], which enables interactively exploring them. Nguyen et al. showed "API usage code adaptation framework" to guide API usage adaptation by learning the API usage adaptation patterns which appeared in other clients who had already migrated to the new library [80]. They argued that the proposed framework compensates for the drawbacks that the existing studies have shown include; In *CatchUp* [81], the library maintainer and application developers should be in the same development environment. In [60], [82], the used modeling technique is too simple. The input of framework incorporates the current version of client application, old and new versions of library, and a set of programs that already migrated to the new library version. This framework is composed of four constituents: *OAT* (origin analysis tool), *CUE* (client API usage extractor), SAM (API usage adaptation miner), and *LIBSYNC*. *LIBSYNC* has a knowledge base of API usage adaptation patterns for each library version, and it detects the locations of client's API usage, which are related to the changed APIs, for the given client system and library version to migrate. And then, it relates each usage with the best suitable API usage pattern in its knowledge base and suggests the edit operations for adaptation.

Some studies have focused on classifying changes and detecting change patterns. Purushothaman and Perry analyzed the impact of small changes, especially one-line changes, about faults, relations between changes (add, delete, and modify), reasons of changes (corrective, adaptive, and perfective), and dependencies between changes [83], [84]. They derived some empirical results as follows: About 10% changes were one-line changes; 50% changes were at most about 10 loc (line of codes) changes;

95% of changes were 50 loc changes; Most changes were 'adaptive' and related to 'code addition'.; Only 4% of one-line changes lead to defects. Zimmermann et al. identified the co-occurring changes in SW systems using changes of entities and association rule mining technique [85]. One example of co-occurring changes is that the modification of function A results in the modification of B and C. They extended their approach to include addition and deletion in [86]. Ying et al. suggested a technique to get change patterns based on data mining and evaluated their approach with Eclipse and Mozilla using predictability and interestingness of the developers [160]. Change patterns in [160] indicate that the set of related files are possible to be changed together from the change history of codes. They insisted that the change patterns help developers notice to the change-related files when they change a file. After preprocessing he extracted the data from SCM system, the association rule mining algorithm was applied, and then, the change patterns were generated and shown via query. However, it is hard to apply Ying et al's approach when the number of transactions is small, and it is also difficult to measure interestingness. Kim et al. conducted fine-grained analysis on function-signature change [87]. The frequency and common patterns of function-signature changes, the frequency distribution of the patterns are uncovered through the analysis. Rysselberghe and Demeyer studied FAC (frequently occurring changes) [88]. Every *CVS* delta is examined via the *CVS* log command, and the corresponding source code changes are recorded in a text file. FAC is the *CVS* deltas, the clones detected by a CCFinder [89]. FAC is regarded as an indicator of the reasons for code redundancy, possible design enhancement, etc., and it helps to identify recurring change patterns and refactoring. Kim et al. studied the change of micro patterns, programming idioms, in JAVA [90]. They focused on the change analysis of class micro pattern types and tried to correlate reported bugs and the change of micro patterns. Three open source projects, *JEdit*, *ArgoUML*, and *Columbia* were selected for the experiment, and Kim et al. concluded that the correlation between changes and bugs remains inconclusive.

To support development and management

Information like individual developer's contribution and expertise of developers is helpful to managers or developers. Huang and Liu grouped developers using logs (deltas) stored in *CVS* repository, and determined the contribution of each developer on the module-level [91]. In the approach, a graph is created, which is composed of nodes and edges indicating developers and 'common contribution relationship', respectively. That is, an edge among developers means they contributed the same directory (module). Casebolt et al. characterized each author's contribution per file using author entropy [55]. Author entropy is based on the entropy theory indicating disorder, and it can estimate the distribution of each author contribution in a file. Their methodology has been applied to *GNOME* project and several observations were presented: When two authors contributed to a file, it is highly possible for large files to have dominant author. Small authors mainly contribute to white space formatting changes, output message changes, interface modifications, and possible bug fixes. Robles et al. proposed a quantitative methodology to study the evolution of core team [93]. In each period, the most active developers are notified and their activities are calculated. Gousious et al. suggested a precise developer contribution measurement by combining traditional contribution metrics and mined data in repositories [94]. The proposed metric is defined with loc of each developer and CF (contribution factor) function per developer. CF is the core in the study, which analyzes the developers' actions into positive and negative, and it sets weights to each action. This metric is theoretically validated in Kaner and Bond metric evaluation framework [95]; however, it is not empirically validated. The credibility of a developer is important in OSS (open source software), as the development team is open to the external developers in many OSS community and the core team in the project wants to involve credible developers to the project. Sinha et al. empirically constructed three hypotheses for increasing the developer's credibility and validated them with Eclipse system [154]. They hypothesized that a developer's credibility is deeply related to his (her) contribution of bugs (H1), his (her) project experience (H2), and the organization which a developer belongs to (H3). In [154], H1 is the most applicable (51%), H3 is the next (38%). They expect that the results are applied to recruit new developers.

Alonso et al. showed a technique to identify and visualize the expertise of a committer with *CVS* data for large open source projects [96]. To do this, the directories of source codes are used as a classification scheme, and transactions are classified according to categories. The size of name is proportional to the number of each committer's transac-

Table 10 Existing approaches to support development and management (part).

task	[ref]	data source	output	target system
characterizing each author's contribution	[55]	SVN	inverse relationship between author entropy and file size	multiple GNOME projects
studying credability of developers	[154]	CVS, SVN, BugZilla	empirical results: corelations between credability and three factors (bug/experience/ organization)	multiple Eclipse projects
identifying expertise based on usage expertise	[97]	CVS	measure for expertise	Eclipse
recommending appropriate artifacts	[98] [99]	CVS, BugZilla, emails	Hipikat: a tool to recommend suitable artifacts in the group memory	Eclipse
supporting software history exploration	[128]	CVS, BugZilla	Rationalizer: a tool to integrate historical information and show the data in view of 'when/who/why'	Eclipse Graphical Editing Framework

tions in the visualization, and the expertise of the committer is also shown. Schuler and Zimmermann recommended developer's expertise based on usage expertise [97]. Agile environment requires dynamic team composition. Existing techniques are mostly based on line 10 rules, that is, the developer who changed the source code most frequently is considered to have expertise. However, they adopted usage expertise, where the developer using functionality via API call is focused. By applying the approach to *Eclipse*, it is shown that experts of a file can be recommended without the help of history data, developers who have similar expertise are the identifier, and API usage can be measured. Minto and Murphy suggested EEL (Emergent Expertise Locator) approach and the tool which presented a ranked list of the emergent team members for a specific task [149]. When a user selects a file in EEL, list of developers who communicate for the file are displayed. To do this, file dependency matrix and expertise matrix based on file authorship matrix are used. File dependency matrix denotes co-modification between two files, and file authorship matrix shows the frequencies of modifications per file for each developer.

Cubranic et al. developed a tool, *Hipikat* to help new developers [98], [99]. Some artifacts like source codes, email, BRs (bug reports), are stored in project memory, and similarity between artifacts are measured via the vector-based IR method. The relationships between artifacts are also derived by heuristics. The appropriate artifacts in project memory are recommended to developers by querying explicitly or by *Hipikat* automatically. Developers try to co-relate various data such as bug reports, checking message, email archives, etc., in order to understand the characteristics of the target codes [151]. To automate this, Holmes and Begel developed a tool, *Deep Intellisense*, which presents various historical information for a single code element by providing current item view, people view, and event history view [151]. Bradley and Murphy also developed *Rationalier*, a tool to show the history of the source code in an integrated form [128]. They compared their results with *Deep Intellisense* [151] via a comparison model that they had constructed. Rationalize show historical data in view of 'when/who/why' for particular code line. They suggested developing more improved tool which incorporates the advantages of the two tools in future work.

Hindle et al. proposed a technique to automatically extract labeled topic to help software maintenance activity in the form of supervised and semi-unsupervised approach [100]. They used only commit comments and not functional requirements but NFR (nonfunctional requirements) are targeted. Thus, the technique is not project-specific and it has cross-project characteristics. Experiments have been conducted on *MySQL* and *MaxDB*. The experimental results show that projects have different relative interests in NFRs and the maintenance activities are affected by external stimuli, not by time.

To enhance Comprehension

There have been several approaches to enhance the software evolution or understandability of the software. Tu and Godfrey developed *Beagle*, which has an analysis component that performs origin analysis and determining change types such as addition or deletion of entities between versions. [101]. The origin analysis consists of *Bertillonage* analysis and dependency analysis. For each release, loc, CC (cyclomatic complexity), and number of parameters are measured and stored as evolution metric vector, and version similarity is defined based on Euclidian distance between each vector. That is, high similarity indicates that post-release is likely to originate from the preceding release. *Bertillonage* analysis is based on similarity and entity-name matching, which is aimed to determine the type of changes like the addition or deletion of entities. Dependency analysis assumes that the clones caused by move or rename tend to follow the original relationships such as call, called-by, etc. in the previous version. *Beagle* provided two simultaneous views for structural and architectural changes via structural diagram and dependency diagram, individually. Structural diagram shows hierarchical view of software entities like subsystems, modules, and functions. Davies et al. proposed a technique to find the origin of software entity using anchored signature matching based on Bertillonage analysis [156]. In this technique, Bertillonage metric for JAVA archive has been defined to match binary class file to source file. The experimental results present that the metric effectively reduces a search space of the candidate source. The dependency diagram presents architectural differences between two releases. Görg and Weiβgerber presented the structural refactoring and local refactoring [102] based on the technique in [103]. Structural refactorings include "move class, move method, pull up method, push down method, and so on". Several refactorings like "hide method, rename method, add/remove parameter" are local refactorings. In [102], class-hierarchy view and package-layout views are given. The various refactorings are distinct with different colors.

Word frequency is used to improve component understanding [104] and to split the identifier for analysis [105]. Kuhn suggested the lexical approach to automatically label the SW component by using log-likelihood ration of word

Table 11 Existing approaches to enhance comprehension (part).

task	[ref]	data source	output	target system
recovery of the origin of entities	[156]	Maven2 central repository	a metric to measure the similarity of two entities	an e-commerce application
splitting identifiers	[105]	SourceForge	Samurai approach: to automatically split identifiers using a scoring technique based on word frequencies	open source Java programs
investigating identifier renamings and their effects	[153]	CVS, SVN	empirical results: several characteristics about identifier renaming	Eclipse-JDT, Tomcat
a change aware environment	[106]	n/a	OperationRecorder: a tool to record editing operations	a reversi game as a Java applet

frequencies and applied it to detect the evolution trends of SW system [104]. Identifiers play an important role to understand programs, because they contain meaningful information such as the intension of the developers [153]. Identifier splitting is required for the analysis of identifiers; however, it is not sufficient to depend only on naming conventions to properly split the identifiers in the source codes. Enslen et al. proposed the *Samurai* approach to automatically split an identifier into words using a scoring technique based on the word frequencies of the source codes [105]. This approach has been applied to nearly 8000 identifiers in the open source JAVA programs. The results show that the proposed approach performs more efficiently than the existing state-of-the-art approaches. Eshkevari et al. investigated identifier renaming (synonym, hypernym, hyponym, and antonym) and studied their effect on the program understanding [153]. They applied their approach to Tomcat and Eclipse-JDT and they found several observations: Renaming occurred frequently during a specific time frame by a part of the developers at class interfaces. The types of renaming include not only the synonym, but also antonym or meronym, which is possible to be error modifications. Many error modifications were conducted on short strings, for example, prefix/suffix change or typo modification. In conclusion, they argued that renaming reflects the changes of the domain model by the developers.

Omori and Maruyama presented a mechanism to record all edit operations conducted on source codes in IDE by a developer, as to enhance the comprehension of the program [106]. It is not sufficient to present individual changes using only current snapshots or the difference between subsequent snapshots, which were used in the existing approaches.

To empirically validate novel ideas and techniques

Several studies have been conducted to empirically validate existing observations or techniques about software evolution. Capiluppi et al. studied the complexity of software systems [107], as to test several hypotheses on the evolutionary characteristics of open sources. Examples of the characteristics are as follows: "As release grows, the functional size becomes larger.", "the potential co-relationship between new developer arrival rate and code growth." The case study on *ARLA* system shows that the number of files and folders grows linearly, and the size is stabilized over release.

Table 12 Existing approaches to empirically validate novel ideas and techniques (part).

task	[ref]	data source	output	target system
evaluating an efficiency metric	[157]	database of customer problem reports	empirical results: characteristic of "mean time to close problem reports"	defect reports (one of IBM's software divisions)
understanding code clones	[110]	CVS	empirical results: characteristics of clones	ArgoUML, DNSJava
validating negativeness of clones	[112]	Git	empirical results: characteristics of clones	Apache, Gimp

The results also indicate that the structure depth becomes nearly constant; however, the width tends to be similar to the number of folders. Those indicate that *ARLA* is a well-structured system. Nugroho et al. evaluated the applicability of *UML* design metrics as to predict the fault-proneness of JAVA classes [45]. They constructed prediction models based on *UML* using the historical data of an industrial JAVA system, and validated it. As a result, it was found that messages from sequence diagrams and the detail level of import coupling can be applied as an important predictor of class fault-proneness, and the precision of the model using *UML* design metrics is higher than the model using code metrics. Zeltyn et al. evaluated "mean time to close problem" which is widely used to measure the efficiency of software maintenance through the accumulated customers' problem reports in IBM [157]. They insisted that 'percentile' is more suitable to measure efficiency than 'mean' for handling time.

Vetro et al. studied the capability of *FindBug*, which is a popular bug finding tool [108]. They applied it to their university java projects, and only two issues out of fifteen issues had high precision. They argued that the technique they used in [108] helps to reduce the information overload of developers.

The characteristics of clones have been empirically investigated. Kim and Notkin suggested an approach based on clone's history to support maintenance [109]. In their approach, a directed graph is generated that consists of nodes indicating clone groups per versions and edges indicating the relations between the groups. A set of clone lineage which originated from the same clone group becomes clone genealogy. Based on the clone genealogy, some research questions are investigated: "how many do source clones require significant maintenance challenge?", "Is aggressive refactoring is the best solution for clone maintenance?." They concluded that clones should be maintained, not removed during evolution. There is no consensus on the consistency of clones. To the contrary, Aversano et al. insisted that the clone groups change consistently [110]. Krinke showed that about 50% of clones change consistently and the rate of consistent change does not increase as version grows [111]. Lozano et al. developed CloneTracker, a tool which detects the rate of change of applications containing clones, and they applied it to DnsJAVA [150]. They found out that the cloned codes were more changeable; however, they concluded that their foundation cannot be generalized, because DnsJAVA had been developed by only two developers. After that, they observed that clones influenced the maintenance effort by analyzing the effect of clones to changeability. However, they could not find systematic relation between them [159]. Rahman et al. empirically validated the general negative characteristics of clones [112]. The relations between clones and defect-proneness have been analyzed, and they concluded that most of bugs are not seriously related to clones, clones are less defect-prone than non-cloned codes. They also presented that there is little evidence that frequently copied clones are more error-prone. In other words, they asserted that clones are not "bed

smell".

Bachmann and Bernstein explored the extent the bug fixing process is affected by the process data quality and characteristics, and what influence does the process quality measured with process data have on product quality [113]. Six open sources and two closed sources have been selected for an empirical study, and the results showed that the quality and characteristics of process data have an effect on the bug-fixing process. For example, the ratio of empty commit has been related with the bug report quality, in *Eclipse*. They also presented that the product quality that has been measured with the number of bug reports was affected by the quality of process data. They noted that those results are applicable to enhance the process quality and product quality. Boogerd and Moonen tried to clarify the correlations between observing coding standard and fault introduction [114]. Several aspects for violation and faults have been investigated: "Is release/file/modules with higher violation density more fault-prone?" and "Is violated lines more faulty?". Cross-release, in-release and line-based analyses were conducted to reveal the aspects. In the cross-release analysis, violations and faults are not related. However, they showed that the ten rules of coding standard can be significant predictors in the in-release analysis and line-based analysis.

To understand software development and software evolution

This category introduces existing works trying to understand the derived characteristics or trends of software development and evolution through empirical study.

Maalej and Happel explored the way software developers describe their jobs [118]. For eight years, they analyzed 75,000 work descriptions of 2,000 professionals and found that there are similarities between metadata of contents and time in the description; the typical pattern is "ACTION concerning ARTIFACT because of CAUSE" [118]. Developers did not describe their job in detail. They argued that the result is applicable to automatically generate the work diaries of developers. Hindle et al. tried to understand large commits which include a large number of files [119]. The large commits had been not usually considered in MSR [119]. They manually classified large commits in 9 open sources and compared them with small commits. From their observations, large commits tend to be 'perfective' and small commits tend to more 'corrective' which is about failure handling. 'Perfective' is related to improve efficiency, performance and maintainability. They insisted that large commits provide insight into the method of project development and reflect the development practices of authors. Layman et al. mined software effort data via *VSTS* (Visual Studio Team System) [120]. In statistically analyzing 55 features effort estimation data of *VSTS* 2008 release, they discovered that actual estimation errors were positively correlated with feature size. In addition to it, they found that the in-process metrics of estimation error were related to the final estimation error and the team conversation was helpful in uncovering the cause of effort estimation inaccuracy. In [120], visualization supported to identify the estimation errors.

German presented a methodology to recover project evolution using software trails [115]. Trails include version releases, version control logs, and mailing lists. Based on the trails using *SoftChange* [92], he studied the evolution of *Evolution*, which is the email client, between 1998 and 2003. Here are some of the results: The distribution size grows faster than source code size; a developer tends to concentrate on one module; MR (modification request) includes a small number of files, etc. In other research by German, subsequent changes have been grouped as MR using *CVS* annotations [116]. In the maintenance period mainly executed bug-fix, the number of MRs is smaller than the improvement period when new functionalities are added. Most of files have been modified many times by same developers. Though there are some cases which different developers modify a set of common files, the files usually belong to the same module. Nikora and Munson examined the source of variations in the set of metrics composed of twelve size metrics and CC under evolution [117]. The assumption is that all kinds of changes do not uniformly affect the whole complexity of a system. For example, the change, adding comments, has less impact on module structure than other changes. They verified the assumption and investigated the suitableness of structural metrics to predict faults and the kinds of changes that contribute to insert faults in the system. They concluded that the control structure is changed more rapidly than others and many changes are caused by the control structure change. The change activities had been fluctuated in all domains of few beginning builds; however, after a specific build when control structure domain becomes a dominant factor, they are stabilized. Herraiz et al. empirically described the SOC (self organized criticality) dynamics of libre software [152]. In the previous study [162], Wu insisted that libre software is SOC, that is, current state of project has been already determined before. Extensive experiments have been conducted in [152] and the results present that evolution of libre software is not SOC. Krishnan et al. analyzed the failure trend and the change trend of common/variable components in software product line, and investigated the relationship between fail-

Table 13 Existing approaches to understand software development and software evolution (part).

task	[ref]	data source	output	target system
exploring the ways of descrbing projects	[118]	SVN, source control system	describing pattern	MyComp, Apache, Eureka
understanding software evolution	[115]	CVS, email archives, BugZilla	empirical observations: characteristics of software evolution	Evolution
understanding evolution of software product line	[158]	CVS	empirical observations: failure trend and change trend of commom/ variable component in software product line	Eclipse

ure and change [158]. Eclipse system, regarded as the evolving product line, was selected for experiment. They revealed that the number of serious failures and changes decreased in the common component. However, in the case of variable components, which were used by five or more products, did not show uniformly decreasing patterns. This indicates that more detailed study is needed in the product line area.

6. Evaluation

Most of MSR studies use open source projects, such as KDE, GCC, Apache, *Eclipse*, *JEdit*, or *ArgoUML*, as their experimental repositories [121]. They are often used for evaluating the results or proposed approaches in MSR.

One of the most commonly used metrics for the evaluations are recall and precision. They were used in various MSR projects [65], [69], [75], [86], [99], [108], [122], [123]. For example, the precision of *FindBug*s was calculated based on NI and NA where NI is the number of predicted issues given by *FindBug*s and NA is the number of actual defects. Recall was not defined in their research because they did not get the complete set of actual defects. However, manual checks have been conducted in order to get the number of actual modifications in *UMLDiff* [122]. Precision and recall were also used for detecting API refactoring [75] and sometimes F-measure that is equal to their harmonic average are applied in order to consider the recall and precision at the same time [127]. The accuracy, precision and recall for evaluation of classification or detection problem are usually defined as (TP+TN)/(TP+TN+FP+FN), TP/(TP+FP) and (TP/TP+FN) where TP, TN, FP and FN is true positive, true negative, false positive and false negative, respectively [26], [31], [124].

There are many models for defect prediction; however, evaluation of them is still an 'open question' [165]. D'Ambros et al. classified the types of prediction into classification, ranking, and effort-aware ranking and describes suitable evaluation metrics for each type [167]. Classification presents the results of the prediction into several types, for example, 'defective/non-defective' are derived from binary classification. To evaluate the classification technique, AUC (the area under the ROC curve) is recommended because precession and recall are sensitive to threshold [167]. AUC is scalar performance measure from ROC (the receiver operating characteristics). X-axis of ROC curve is the probability of false alarm, and y-axis is the detection probability. AUC is measured as the area between the curve and x-axis in ROC. Lessmann et al. experimented 10 data sets of NASA MDP (metrics data program) with AUC applying 22 classifier models [164]. They found out that the performance of the classification models was not significantly different and their roles had been overestimated in previous studies [164]. In ranking, the list of ranked modules as results is known as MOM (module-order model) [166]. MOM is a software quality model which is to be used for predicting the rank-order of each module according to the quality factor such as the number of faults [166]. To evaluate MOM, P_{opt} is used based on cumulative lift charts, where x-axis is the ordered modules according to a prediction model and y-axis is a cumulative ratio of the identified defects. P_{opt} suggested by Mende and Koschke [165] is defined as '1-delta(opt)', where delta(opt) is the difference between the prediction model and the optimal model. Effort-aware ranking incorporates ranking and efforts to review. Existing evaluation techniques assume that additional QA (quality assurance) cost is equal for each module, however, the cost to review or test is highly related to size. P_{opt} is also used in evaluating effort-aware ranking, but P_{eff} is used in [167] to avoid confusion. X-axis of the cumulative lift chart is the ordered classes according to defect density. That is, the criteria is simple defect count in ranking, however, in effort-aware ranking, the criteria of X-axis is a value derived by dividing the number of defects with loc of each module. In addition, CE (cost effectiveness) has been defined [165], which is the difference between the random model and prediction model. CE and P_{opt} is similar, but their usage is different; CE provides insight of cost effectiveness for prediction model and P_{opt} is used to fairly evaluate the predictor performance [165]. An information-theoretic approach could be used in order to evaluate the probabilistic models to predict bugs or changes [125]. The effectiveness of the approach could be e valuated by comparing the distribution of the predicted values and actual values. The entropy measurements are typically used for the distribution metrics.

Statistical methods such as correlation or t-test are also important evaluation techniques [113]. Bachmann and Bernstein used tau (τ) rank correlation coefficient in order to evaluate the correlation between process data and product quality [126]. It is considered to be more effective in the case of having outliers, compared to other coefficient metrics such as Spearman or Pearson correlation coefficient. And then, a t-test can be conducted for the significance of the tau correlation value. For machine learning, 10-fold cross-validation is usually used for determining the training set and test set. For performance comparisons among available learning algorithms, the t-test is applied. The size of the training set should not be too small. However, overfitting also should be avoided because it shows low precision for the unknown or test data even if they have high precision for the training data. When analyzing the data, this kind of bias should be eliminated in order to escape from making a wrong decision.

7. Opportunities and Challenges

The rush for software repositories started less than 10 years ago and there are still a lot of gold mines that could provide new insights into the MSR area. As the data of the software projects become larger and more various, new effective approaches for data extraction, process and analysis will be raised as challenging issues. Some MSR open issues are as follows.

Generalization of mining software repositories
Many MSR studies have been individually conducted according to specific repositories or domains and there is a lack of a standardized model of MSR. The common requirements in the MSR process have not been systematically reflected. The MSR approaches are dependent on the types of data sources. Especially, the data schema for each repository is different and it should be designed for each case. It is a time-consuming, repeated and error-prone work. However, the approaches to processing or analyzing data can have common parts and can be identified to be defined as general approaches. Thus, this kind of work can guide the MSR researchers and provide them more systematic approaches for handling the data and artifacts. Common models could be used for analysis steps and they can be obtained from each of the repository specific models. Meta-repository approach like FLOSSMole [180], [182] and FLOSSMetrics [179], [181] are representative cases for the effort. They are similar in that they provide the FLOSS data in an applicable form by extracting and processing, which results in the reduction of burden in collecting data for MSR researchers [183]. FLOSSMetrics focuses on the developer's activity and it mainly utilizes source control system, mailing lists, and bug tracking system, however, FLOSSMole does not focus on obtaining whole data of a given project. It collects various data such as community and team size, from several different repositories and provides them with various formats [183]. We expect those to become the motive power of future MSR research. The target of generalization includes not only the repository, but also models or methodologies recurring in the overall MSR process. The generalization or standardization of the models or methodologies makes it possible to help MSR researcher with the various aspects.

Supporting mining software repository researchers
This issue is related to the first one because most of the practical supports such as integrated research environments or tools could be implemented after the definition of MSR models. As the MSR process requires time and resource, the optimization of the process is needed [169]. Shang et al. showed a framework to support MSR research using MapReduce [170] which is a framework to handle large volume of data [169]. Mockus constructed universal repository by extracting a quantity of data from the public and corporate VCS [168]. Based on the formalized models or methodologies from various MSR experiences, the MSR approaches could be optimized. Their theoretical models or operations can be used for organizing the MSR research concepts. Existing version control systems such as *CVS* and *SVN* store only snapshop, and in [171], [172], tool to store not only the simple log but changes have also been proposed. For example, Syde in [172] records every change. These attempts to extend the functionality of the VCS enables MSR researchers to utilize change in information without encountering much burden in processing change data. As MSR process essentially utilize various project data accumulated during project time, scale and complexity of data grows and the kinds of data source become varied. Thus, tools or environments to manage the complexity are required more and more.

New development domains or data sources
As shown in tables in [161], MSR researchers have been mostly focused on C/C++, Java applications. There have not been many studies related to the web applications. Thus, these new domains can provide other results from previous domains such as Object-oriented environment. Most of the data sources in [161] were CVS or SVN, however, another version control system like Git, new development contexts or tools such as Mylyn or IBM Jazz can provide new data sources. The newly adopted data elements from these data sources can provide new features and enhance the MSR results. Also, [161] shows that many existing MSR researches have been conducted on the open source software, which indicates that more commercial software should be considered in the MSR research for balanced results.

Applications of other domain approaches
Recently, human aspect in empirical software engineering is on the increase [173], and SNA (social network analysis) in MSR is the main case of it [175]. SNA is diversely adopted; Applying SNA to *CVS* [174], mining mailing list [177], graph visualization of developers' network [176], analyzing developer's blog [178], validation of SNA metrics [173]. As social networks are being actively constructed and a variety of media to communicate appear, social network analysis in MSR is challengeable in the future. Various graph-based approaches to other network domains could be applied to the MSR issues [146]. For example, developers who have high impact on the overall development process or other developers could be retrieved by using the concept of prestige, which is used by *PageRank*.

Quality enhancement of results
Repositories become more complex and provide more detailed data. For example, Mylyn can provide fine-grained data entities such as programmers' editing files or selecting menus with time stamps. *IBM Jazz* can also provide full traceability among all software artifacts, which could enhance the quality of the relational data between the source codes and the bug reports. Actually, connecting the various repositories becomes a challenging issue [8]. Reduced noise of the extracted data can also improve the analysis results. Finally, the increased quality of the data and the enhanced algorithms for the MSR could provide better precision and recall.

8. Conclusion

MSR is a young multidisciplinary research area which includes data mining, artificial intelligence and software engineering. The general process of MSR is composed of extracting, processing and analyzing data sources. The funda-

mental techniques, theories and knowledge for accomplishing the phases mainly come from these heterogeneous studies. Thus, MSR is full of new challenges for combining and adapting new approaches to solve interesting issues such as defect or change prediction, bug report classification, developer guidance and so on.

Knowledge and information that can be obtained by mining historical software repositories can improve developers' decision-making processes in the future. That will enable them to do their job more efficiently with the acquired knowledge base, by supplementing their intuition and/or experience.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0005632).

References

[1] Zimmer, http://thomas-zimmermann.com/research accessed Nov. 10, 2011.
[2] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in XML documents," Proc. Int'l Conf. on Data Eng., San Jose, CA, USA, pp.41–52, Feb.-March 2002.
[3] Jazz, http://jazz.net accessed June 10, 2011.
[4] Mylyn, http://www.eclipse.org/mylyn accessed June 10, 2011.
[5] MSR, http://www.msrconf.org accessed June 10, 2011.
[6] P. Tan, M. Steinbach, and V. Kumar, Introduction to data mining, Addison Wesley, USA, 2005.
[7] ISO/IEC, http://pascal.computer.org/sev_display accessed June 10, 2011.
[8] A.E. Hassan, "The road ahead for mining software repositories," Proc. Frontiers of Software Maintenance, pp.48–57, Beijing, China, Sept. 2008.
[9] CVS, http://www.nongnu.org/cvs accessed June 10, 2011.
[10] SVN, http://subversion.tigris.org accessed June 10, 2011.
[11] Git, http://git-scm.com accessed June 10, 2011.
[12] Mercurial, http://mercurial.selenic.com accessed June 10, 2011.
[13] Bazzar, http://bazaar.canonical.com accessed June 10, 2011.
[14] Darcs, http://darcs.net/ accessed June 10, 2011.
[15] S. Kim, K. Pan, and E. James Whitehead Jr., "Memories of bug fixes," Proc. Int'l Symposium on Foundations of Software Engineering, pp.35–45, Graz, Austria, March 2006.
[16] JLint, http://artho.com/jlint accessed June 10, 2011.
[17] FindBugs, http://findbugs.sourceforge.net accessed June 10, 2011.
[18] S. Kim, T. Zimmermann, E.J. Whitehead, Jr., and A. Zeller, "Predicting faults from cached history," Proc. Int'l Conf. on Software Engineering, pp.489–498, Minneapolis, MN, USA, May 2007.
[19] BugZilla, http://www.bugzilla.org accessed June 10, 2011.
[20] Trac, http://trac.edgewall.org accessed June 10, 2011.
[21] JIRA, http://www.atlassian.com/software/jira accessed June 10, 2011.
[22] MySQL, http://www.mysql.com accessed June 10, 2011.
[23] Postgre, http://www.postgresql.org accessed June 10, 2011.
[24] Oracle, http://www.oracle.com accessed June 10, 2011.
[25] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," Proc. Int'l Symposium on Foundations of Software Eng., pp.308–318, Atlanta, GA, USA, Nov. 2008.
[26] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," Proc. Int'l Working Conf. on Mining Software Repositories, pp.27–30, Leipzig, Germany, May 2008.
[27] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... Really?," Proc. Int'l Conf. on Software Maintenance, pp.337–345, Beijing, China, Sept. 2008.
[28] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," Proc. Conf. on Dependable Systems and Networks, pp.52–61, Anchorage, Alaska, USA, June 2008.
[29] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language Processing," Proc. Int'l Conf. on Software Eng., pp.499–510, Minneapolis, MN, USA, May 2007.
[30] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," Proc. Int'l Conf. on Software Eng., pp.461–470, Leipzig, Germany, May 2008.
[31] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," Proc. Working Conf. Mining Software Repositories, pp.1–10, Cape Town, South Africa, May 2010.
[32] L. Yu, W.T. Tsai, W. Zhao, and F. Wu, "Predicting defect priority based on neural networks," Proc. Int'l Conf. on Advanced Data Mining and Applications, pp.356–367, Chongqing, China, Nov. 2010.
[33] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," Proc. Working Conf. on Mining Software Repositories, pp.11–20, Cape Town, South Africa, May 2010.
[34] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," Proc. Int'l Conf. on Software Eng., pp.521–530, Leipzig, Germany, May 2008.
[35] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?," Proc. Int'l Symposium on Foundations of Software Eng., pp.2–12, Atlanta, Georgia, USA, Nov. 2008.
[36] M.E. Conway, "How do committees invent?," Datamation, vol.14, no.5, pp.28–31, 1968.
[37] K.A. Schneider, C. Gutwin, R. Penner, and D. Paquette, "Mining a software developer's local interaction history," Proc. Int'l Workshop on Mining Software Repositories, pp.106–110, Edinburgh, Scotland, UK, May 2004.
[38] L. Yu, S. Ramaswamy, and C. Zhang, "Mining email archives and simulating the dynamics of open-source project developer networks," Proc. Int'l Workshop on Enterprise & Organizational Modeling and Simulation, pp.17–31, Montpellier, France, June 2008.
[39] E. Shihab, Z.M. Jiang, and A.E. Hassan, "On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK+ project," Proc. Int'l Working Conf. on Mining Software Repositories, pp.107–110, Vancouver, Canada, May 2009.
[40] E. Shihab, Z.M. Jiang, and A.E. Hassan, "Studying the use of developer IRC meetings in open source projects," Proc. Int'l Conf. on Software Maintenance, pp.147–156, Edmonton, Alberta, Canada, Sept. 2009.
[41] XMI, http://www.omg.org/spec/XMI accessed June 10, 2011.
[42] Visual-paradigm, http://www.visual-paradigm.com accessed June 10, 2011.
[43] Enterprise Architect, http://www.sparxsystems.com accessed June 10, 2011.
[44] IBM-tau, www.ibm.com/software/awdtools/tau accessed June 10, 2011.
[45] A. Nugroho, M.R.V. Chaudron, and E. Arisholm, "Assessing UML design metrics for predicting fault-prone classes in a Java system," Proc. Int'l Working Conf. on Mining Software Repositories, pp.21–30, Cape Town, South Africa, May 2010.
[46] T.H.D. Nguyen, A. Schröter, and D. Damia, "Mining jazz: An ex-

perience report," Proc. Int'l Workshop on Infrastructure for Research in Collaborative Software Eng., Atlanta, GA, USA, Nov. 2008.
[47] K. Herzig and A. Zeller, "Mining the Jazz repository: Challenges and opportunities," Proc. Int'l Working Conf. on Mining Software Repositories, pp.159–162, Vancouver, Canada, May 2009.
[48] S. Rastkar and G.C. Murphy, "On what basis to recommend: Changesets or interactions?," Proc. Int'l Working Conf. on Mining Software Repositories, pp.155–158, Vancouver, Canada, May 2009.
[49] M. Kersten and G.C. Murphy, "Using task context to improve programmer productivity," Proc. Int'l Symposium on Foundations of Software Engineering, pp.1–11, Portland, OR, USA, Nov. 2006.
[50] ANTLR, http://www.antlr.org accessed June 10, 2011.
[51] JDT, http://www.eclipse.org/jdt accessed June 10, 2011.
[52] Java2XML, http://sourceforge.net/projects/java2xml accessed June 10, 2011.
[53] xmlDiff, http://diffxml.sourceforge.net/ accessed June 10, 2011.
[54] Y. Wang, D.J. Dewitt, and J.Y. Cai, "X-Diff: An effective change detection algorithm for XML documents," Proc. Int'l Conf. on Data Eng., pp.519–530, Bangalore, India, March 2003.
[55] J.R. Casebolt, J.L. Krein, A.C. MacLean, C.D. Knutson, and D.P. Delorey, "Author entropy vs. file size in the gnome suite of applications," Proc. Int'l Working Conf. on Mining Software Repositories, pp.91–94, Vancouver, Canada, May 2009.
[56] A.E. Hassan, "Predicting faults using the complexity of code changes," Proc. Int'l Conf. on Software Eng., pp.78–88, Vancouver, Canada, May 2009.
[57] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," Proc. Int'l Conf. on Software Eng., Waikiki, Honolulu, Hawaii, May 2011.
[58] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," Proc. Int'l Workshop on Mining Software Repositories, pp.24–28, St. Louis, MO, USA, May 2005.
[59] C.C. Williams and J.K. Hollingsworth, "Bug driven bug finders," Proc. Int'l Workshop on Mining Software Repositories, pp.70–74, Edinburgh, Scotland, UK, May 2004.
[60] C.C. Williams and J.K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," IEEE Trans. Softw. Eng., vol.31, no.6, pp.466–480, June 2005.
[61] S. Sudhakrishnan, J.T. Madhavan, E.J. Whitehead Jr., and J. Renau, "Understanding bug fix patterns in Verilog," Proc. Int'l Working Conf. on Mining Software Repositories, pp.39–42, Leipzig, Germany, May 2008.
[62] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," Proc. Int'l Conf. on Software Eng., pp.452–461, Shanghai, China, May 2006.
[63] J. Ekanayake, J. Tappolet, H.C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," Proc. Int'l Working Conf. on Mining Software Repositories, pp.51–60, Vancouver, Canada, May 2009.
[64] E. Giger, M. Pinzger, and H.C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," Proc. Int'l Working Conf. on Mining Software Repositories, pp.83–92, Honolulu, HI, May 2011.
[65] J. Anvik, L. Hiew, and G.C. Murphy, "Who should fix this bug?," Proc. Int'l Conf. on Software Eng., pp.361–370, Shanghai, China, May 2006.
[66] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based experties model of developers," Proc. Int'l Working Conf. on Mining Software Repositories, pp.131–140, Vancouver, Canada, May 2009.
[67] V. Dallmeier and T. Zimmermann, "Automatic extraction of bug localization benchmarks from history," Proc. Int'l Conf. on Automated Software Eng., pp.433–436, Atlanta, GA, USA, Nov. 2007.
[68] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," Proc. Int'l Working Conf. on Mining Software Repositories, pp.43–52, Honolulu, HI, May 2011.
[69] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," Proc. Int'l Symposium on Software Metrics, pp.29–37, Como, Italy, Sept. 2005.
[70] G. Canfora and L. Cerulo, "Fine grained indexing of software repositories to support impact analysis," Proc. Int'l Workshop on Mining Software Repositories, pp.105–111, Shanghai, China, May 2006.
[71] R. Robbes, D. Pollet, and M. Lanza, "Replaying IDE interactions to evaluate and improve change prediction approaches," Proc. Int'l Working Conf. on Mining Software Repositories, pp.161–170, Cape Town, South Africa, May 2010.
[72] P. Weißgerber and S. Diehl, "Are refactorings less error-prone than other changes?," Proc. Int'l Workshop on Mining Software Repositories, pp.112–118, Shanghai, China, May 2006.
[73] J. Ratzinger, T. Sigmund, and H.C. Gall, "On the relation of refactoring and software defects," Proc. Int'l Working Conf. on Mining Software Repositories, pp.35–38, Leipzig, Germany, May 2008.
[74] D. Dig, and R. Johnson, "How do APIs evolve? A story of refactoring," Journal of Software Maintenance and Evolution: Research and Practice, vol.18, no.2, pp.83–107, 2006.
[75] K. Taneja, D. Dig, and T. Xie, "Automated detection of api refactorings in libraries," Proc. Int'l Conf. on Automated Software Eng., pp.377–380, Atlanta, GA, USA, Nov. 2007.
[76] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," Proc. European Conf. on Object-Oriented Programming, pp.404–428, Nantes, France, July 2006.
[77] Z. Xing and E. Stroulia, "API-Evolution support with diff-catchup," IEEE Trans. Softw. Eng., vol.33, no.12, pp.818–836, Dec. 2007.
[78] Z. Xing and E. Stroulia, "Differencing logical UML models," J. Automated Software Eng., vol.14, no.2, pp.215–259, June 2007.
[79] Z. Xing and E. Stroulia, "Bottom-up design evolution concern discovery and analysis," Technical Report, TR07-13, Univ. of Alberta, July 2007.
[80] H.A. Nguyen, T.T. Nguyen, G. Wilson Jr., A.T. Nguyen, M. Kim, and T.N. Ngyuen, "A Graph-based approach to API usage adaptation," Proc. Int'l Conf. on Object-oriented programming systems languages and applications, pp.302–321, Reno/Tahoe, NV, USA, Oct. 2010.
[81] J. Henkel and A. Diwan, "CatchUp!: Capturing and replaying refactorings to support API evolution," Proc. Int'l Conf. on Software Eng., pp.274–283, St. Louis, MO, USA, May 2005.
[82] B. Dagenais and M.P. Robillard, "SemDiff: Analysis and recommendation support for API evolution," Proc. Int'l Conf. on Software Eng., pp.599–602, Vancouver, Canada, May 2009.
[83] R. Purushothaman and D.E. Perry, "Towards understanding the rhetoric of small changes," Proc. Int'l Workshop on Mining Software Repositories, pp.90–94, Edinburgh, Scotland, UK, May 2004.
[84] R. Purushothaman and D.E. Perry, "Toward understanding the rhetoric of small source code changes," IEEE Trans. Softw. Eng., vol.31, no.6, pp.511–526, 2005.
[85] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," Proc. Int'l Conf. on Software Eng., pp.563–572, Edinburgh, Scotland, UK, May 2004.
[86] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl, "Mining version histories to guide software changes," IEEE Trans. Softw. Eng., vol.31, no.6, pp.429–445, 2005.
[87] S. Kim, E.J. Whitehead, and J. Bevan, "Analysis of signature change patterns," Proc. Int'l Workshop on Mining Software Repositories, pp.64–68, St. Louis, MO, USA, May 2005.
[88] F. Van Rysselberghe and S. Demeyer, "Mining version control systems for FACs (frequently applied changes)," Proc. Int'l Workshop on Mining Software Repositories, pp.48–52, Edinburgh, Scotland,

UK, May 2004.
[89] CCFinder, http://www.ccfinder.net accessed June 10, 2011.
[90] S. Kim, K. Pan K, and E.J. Whitehead Jr., "Micro pattern evolution," Proc. Int'l Workshop on Mining Software Repositories, pp.40–46, Shanghai, China, May 2006.
[91] S.K. Huang and K.M. Liu, "Mining version histories to verify the learning process of legitimate peripheral participants," Proc. Int'l Workshop on Mining Software Repositories, pp.84–78, St. Louis, MO, USA, May 2005.
[92] D.M. German, "Mining CVS repositories, the softChange experience," Proc. Int'l Workshop on Mining Software Repositories, pp.17–21, Edinburgh, Scotland, UK, May 2004.
[93] G. Robles, J.M. Conzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in Libre software projects," Proc. Int'l Working Conf. on Mining Software Repositories, pp.167–170, Vancouver, Canada, May 2009.
[94] G. Gousios, E. Kalliamvakou, and D. Spinellis, "Measuring developer contribution from source repository data," Proc. Int'l Working Conf. on Mining Software Repositories, pp.129–132, Leipzig, Germany, May 2008.
[95] C. Kaner and W. Bond, "Software engineering metrics: What do they measure and how do we know?," Proc. Int'l Software Metrics Symposium, pp.1–12, Chicago, IL, USA, Sept. 2004.
[96] O. Alonso, P.T. Devanbu, and M. Gertz, "Expertise Identification and Visualization from CVS," Proc. Int'l Working Conf. on Mining Software Repositories, pp.125–128, Leipzig, Germany, May 2008.
[97] D. Schuler and T. Zimmermann, "Mining Usage Expertise from Version Archives," Proc. Int'l Working Conf. on Mining Software Repositories, pp.121–124, Leipzig, Germany, May 2008.
[98] D. Cubranic and G.C. Murphy, "Hipikat: Recommending pertinent software development artifacts," Proc. Int'l Conf. on Software Eng., pp.408–418, Portland, OR, USA, May 2003.
[99] D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth, "Hipikat: A project memory for software development," IEEE Trans. Softw. Eng., vol.31, no.6, pp.446–465, 2005.
[100] A. Hindle, N.A. Ernst, M.W. Godfrey, and J. Mylopoulos, "Automated Topic naming to support cross-project analysis of software maintenance activities," Proc. Int'l Working Conf. on Mining Software Repositories, pp.163–172, Honolulu, HI, May 2011.
[101] Q. Tu and M.W. Godfrey, "An integrated approach for studying architectural evolution," Proc. Int'l Workshop on Program Comprehension, pp.127–136, Paris, France, June 2002.
[102] C. Görg and P. Weiβgerber, "Detecting and visualizing refactorings from software archives," Proc. Int'l Workshop on Program Comprehension, pp.205–214, St. Louis, MO, USA, May 2005.
[103] C. Görg and P. Weiβgerber, "Error detection by refactoring reconstruction," Proc. Int'l Workshop on Mining Software Repositories, pp.29–33, St. Louis, MO, USA, May 2005.
[104] A. Kuhn, "Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code," Proc. Int'l Working Conf. on Mining Software Repositories, pp.175–178, Vancouver, Canada, May 2009.
[105] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining Source Code to Automatically Split Identifiers for Software Analysis," Proc. Int'l Working Conf. on Mining Software Repositories, pp.71–80, Vancouver, Canada, May 2009.
[106] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," Proc. Int'l Working Conf. on Mining Software Repositories, pp.31–34, Leipzig, Germany, May 2008.
[107] A. Capluppi, M. Moriso, and J.F. Ramil, "Structural evolution of an open source system: A case study," Proc. Int'l Workshop on Program Comprehension, pp.172–182, Bari, Italy, June 2004.
[108] A. Vetro, M. Torchiano, and M. Morisio, "Assessing the precision of FindBugs by mining Java projects developed at a university," Proc. Int'l Working Conf. on Mining Software Repositories, pp.110–113, Cape Town, South Africa, May 2010.
[109] M. Kim and D. Notkin, "Using a clone genealogy extractor for understanding and supporting evolution of code clones," Proc. Int'l Workshop on Mining Software Repositories, pp.17–21, St. Louis, MO, USA, May 2005.
[110] L. Aversano, L. Cerulo, and M.D. Penta, "How clones are maintained: An empirical study," Proc. European Conf. on Software Maintenance and Reengineering, pp.81–90, Amsterdam, Netherlands, March 2007.
[111] J. Krinke, "A study of consistent and inconsistent changes to code clones," Proc. Working Conf. on Reverse Eng., pp.170–178, Vancouver, Canada, Oct. 2007.
[112] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?," Proc. Int'l Working Conf. on Mining Software Repositories, pp.72–81, Cape Town, South Africa, May 2010.
[113] A. Bachmann and A. Bernstein, "When process data quality affects the number of bugs: Correlations in software engineering datasets," Proc. Int'l Working Conf. on Mining Software Repositories, pp.62–71, Cape Town, South Africa, May 2010.
[114] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faults within and across software versions," Proc. Int'l Working Conf. on Mining Software Repositories, pp.41–50, Vancouver, Canada, May 2009.
[115] D.M. German, "Using software trails to reconstruct the evolution of software," J. Software Maintenance and Evolution: Research and Practice, vol.16, no.6, pp.367–384, 2004.
[116] D.M. German, "An empirical study of fine-grained software modifications," Proc. Int'l Conf. on Software Maintenance, pp.316–325, Chicago, IL, USA, 2004.
[117] A.P. Nikora and J.C. Munson, "Understanding the nature of software evolution," Proc. Int'l Conf. on Software Maintenance, pp.83–93, Amsterdam, Netherlands, Sept. 2003.
[118] W. Maalej and H. Happel, "From work to word: How do software developers describe their work?," Proc. Int'l Working Conf. on Mining Software Repositories, pp.121–120, Vancouver, Canada, May 2009.
[119] A. Hindle, D.M. German, and R. Holt, "What do large commits tell us? A taxonomical study of large commits," Proc. Int'l Working Conf. on Mining Software Repositories, pp.99–108, Leipzig, Germany, May 2008.
[120] L. Layman, N. Nagappan, S. Guckenheimer, and J. Beehler, "Mining software effort data: Preliminary analysis of visual studio team system data," Proc. Int'l Working Conf. on Mining Software Repositories, pp.43–46, Leipzig, Germany, May 2008.
[121] H. Kagdi, M.L. Collard, and J.I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," J. Software Maintenance and Evolution: Research and Practice, vol.19, no.2, pp.77–131, March 2007.
[122] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," Proc. Int'l Conf. on Automated Software Eng., pp.54–65, Long Beach, CA, USA, Nov. 2005.
[123] A.E. Hassan and R.C. Holt, "Predicting change propagation in software systems," Proc. Int'l Conf. on Software Maintenance, pp.284–293, Chicago, IL, USA, Sept. 2004.
[124] G. Canfora, C. Luigi, and D.P. Massimiliano, "Identifying changed source code lines from version repositories," Proc. Int'l Workshop on Mining Software Repositories, p.14, Minneapolis, USA, May 2007.
[125] M. Askari and R. Holt, "Information theoretic evaluation of change prediction models for large-scale software," Proc. Int'l Workshop on Mining Software Repositories, pp.126–132, Shanghai, China, May 2006.
[126] M.G. Kendall, "A new measure of rank correlation," Biometrika, vol.30, no.1/2, pp.81–93, June 1938.
[127] C.J. van Rijsbergen, "Information Retrieval," Butterworth, 1979.
[128] A.W.J. Bradley and G.C. Murphy, "Supporting software history exploration," Proc. Int'l Working Conf. on Mining Software Repositories, pp.193–202, Honolulu, HI, May 2011.

[129] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," Proc. Working Conf. on Mining Software Repositories, pp.11–20, Cape Town, South Africa, May 2010.

[130] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," IEEE Trans. Softw. Eng., vol.31, no.10, pp.897–910, Oct. 2005.

[131] S. Kim, T. Zimmermann, K. Pan, and E.J. Jr. Whitehead, "Automatic identification of bug-introducing changes," Proc. Automated Software Engineering, pp.81–90, Tokyo, Japan, Sept. 2006.

[132] J. Anvik and G.C. Murphy, "Determining implementation experties from bug reports," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[133] F. Khomh, B. Chan, Y. Zou, and A.E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of Mozilla Firefox," Proc. Working Conf. on Reverse Eng., Lero, Ireland, Oct. 2011.

[134] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen, "Mining software reporitories to study co-evolution of production & test code," Proc. Int'l Conf. on Software Testing, Verification, and Validation, pp.220–229, Lillehammer, Norway, April 2008.

[135] J. Long, "Understanding the role of core developers in open source development," J. Informationm, Information Technology, and Organizations, vol.1, pp.75–85, 2006.

[136] A.E. Hassan and T. Xie, "Software intelligence: The future of mining software engineering data," Proc. Working Conf. on Future of Software Eng., pp.161–166, Santa Fe, NM, USA, Nov. 2010.

[137] G. Robles, "Empirical software engineering research on libre software: Data sources, methodologies and results," Doctorial Thesis, Universidad Rey Juan Carlos, 2006. (http://libresoft.es/publications/thesis-grex)

[138] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. German, and P.T. Devanbu, "The promise and perils of mining git," Proc. Int'l Working Conf. on Mining Software Repositories, pp.175–178, Vancouver, Canada, May 2009.

[139] P.C. Rigby and A.E. Hassan, "What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[140] S. Zaman, B. Adams, and A.E. Hassan, "Security versus performance bugs: A case study on FireFox," Proc. Int'l Working Conf. on Mining Software Repositories, pp.93–102, Honolulu, HI, May 2011.

[141] D.M. German, "Using software distributions to understanding the relationship among free and open source software projects," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[142] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno, "FAVE: Factor analysis based approach for detecting product line variability from change history," Proc. Int'l Working Conf. on Mining Software Repositories, pp.11–18, Leipzig, Germany, May 2008.

[143] W. Maalej and H. Happel, "Can development work describe itself?," Proc. Int'l Working Conf. on Mining Software Repositories, pp.191–200, Cape Town, South Africa, May 2010.

[144] A. Mauczka, C. Schanes, F. Fankhauser, M. Bernhart, and T. Grechenig, "Mining security changes in FreeBSD," Proc. Int'l Working Conf. on Mining Software Repositories, pp.90–93, Cape Town, South Africa, May 2010.

[145] W.M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A.E. Hassan, "Should I contribute to this discussion?," Proc. Int'l Working Conf. on Mining Software Repositories, pp.181–190, Cape Town, South Africa, May 2010.

[146] M. Halkidi, D. Spinellis, G. Tsatsaronis, and M. Vazirgiannis, "Data mining in software engineering," Intelligent Data Analysis, vol.15, no.3, pp.413–441, May 2011.

[147] S. Morisaki, A. Monden, T. Matsumura, H. Tamada, and K. Matsumoto, "Defect data analysis based on extended association rule mining," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[148] L. Yu and S. Ramaswamy, "Mining CVS repositories to understand open-source project developer roles," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[149] S. Minto and G.C. Murphy, "Recommending emergent teams," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[150] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[151] R. Holmes and A. Begel, "Deep Intellisense: A tool for rehydrating evaporated information," Proc. Int'l Working Conf. on Mining Software Repositories, pp.23–26, Leipzig, Germany, May 2008.

[152] I. Herraiz, J.M. Gonzalez-Barahona, and G. Robles, "Determinism and evolution," Proc. Int'l Working Conf. on Mining Software Repositories, pp.1–10, Leipzig, Germany, May 2008.

[153] L.M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y. Guéhéneuc, and G. Antoniol, "An exploratory study of identifier renamings," Proc. Int'l Working Conf. on Mining Software Repositories, pp.33–42, Honolulu, HI, May 2011.

[154] V.S. Sinha, S. Mani, and S. Sinha, "Entering the circle of trust: Developer initiation as committers in open-source projects," Proc. Int'l Working Conf. on Mining Software Repositories, pp.133–142, Honolulu, HI, May 2011.

[155] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess," Proc. Int'l Working Conf. on Mining Software Repositories, pp.143–152, Honolulu, HI, May 2011.

[156] J. Davies, D.M. German, M.W. Godfrey, and A. Hindle, "Software bertillonage: Finding the provenance of an entity," Proc. Int'l Working Conf. on Mining Software Repositories, pp.183–192, Honolulu, HI, May 2011.

[157] S. Zeltyn, P. Tarr, M. Cantor, R. Delmonico, S. Kannegala, M. Keren, A.P. Kumar, and S. Wasserkrug, "Improving efficiency in software maintenance," Proc. Int'l Working Conf. on Mining Software Repositories, pp.215–218, Honolulu, HI, May 2011.

[158] S. Krishnan, R.R. Lutz, and K. Goseva-Popstojanova, "Empirical evaluation of reliability improvement in an evolving software product-line," Proc. Int'l Working Conf. on Mining Software Repositories, pp.113–122, Honolulu, HI, May 2011.

[159] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," Proc. Int'l Conf. on Software Maintenance, pp.227–236, Beijing, China, Sept.-Oct. 2008.

[160] T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting source code changes by mining change history," IEEE Trans. Softw. Eng., vol.30, no.9, pp.574–586, Sept. 2004.

[161] Summary of MSR purposes, http://zorba.knu.ac.kr/research/MSR_Survey/overall_table.html accessed Dec. 6, 2011.

[162] J. Wu, Open source software evolution and its dynamics, PhD Thesis, University Waterloo, 2006.

[163] C. Weiβ, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.

[164] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," IEEE Trans. Softw. Eng., vol.34, no.4, pp.485–496, July-Aug. 2008.

[165] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction," Proc. Int'l Conf. on Predictor Models in Software Eng., pp.1–10, Vancouver, Canada, May 2009.

[166] T.M. Khoshgoftaar and E.B. Allen, "Ordering fault-prone software modules," Software Quality J., vol.11, no.1, pp.29–37, May 2003.

[167] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison,"

Empir Software Eng., pp.1–47, Aug. 2011.
[168] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," Proc. Int'l Working Conf. on Mining Software Repositories, pp.11–20, Vancouver, Canada, May 2009.
[169] W. Shang, Z.M. Jiang, B. Adams, and A.E. Hassan, "MapReduce as a general framework to support research in Mining Software," Proc. Int'l Working Conf. on Mining Software Repositories, pp.21–30, Vancouver, Canada, May 2009.
[170] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun. ACM, vol.51, no.1, pp.1–13, Jan. 2008.
[171] R. Robbes, "Mining a change-based software repository," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.
[172] L. Hattori and M. Lanza, "Mining the history of synchronous changes to refine code ownership," Proc. Int'l Working Conf. on Mining Software Repositories, pp.175–178, Vancouver, Canada, May 2009.
[173] R. Nia, C. Bird, P.T. Devanbu, and V. Filkov, "Validity of network analyses in open source projects," Proc. Int'l Working Conf. on Mining Software Repositories, pp.201–209, Cape Town, South Africa, May 2010.
[174] L. Lopez-Fernandez, G. Robles, and J.M. Gonzalez-Barahona, "Applying social network analysis to the information in CVS repositories," Proc. Int'l Workshop on Mining Software Repositories, pp.101–105, Edinburgh, Scotland, UK, May 2004.
[175] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, "Using software repositories to investigate socio-technical congruence in development projects," Proc. Int'l Workshop on Mining Software Repositories, Minneapolis, USA, May 2007.
[176] B. Heller, E. Marschner, E. Rosenfeld, and J. Heer, "Visualizing collaboration and infuluence in open-source software community," Proc. Int'l Working Conf. on Mining Software Repositories, pp.223–226, Honolulu, HI, May 2011.
[177] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," Proc. Int'l Workshop on Mining Software Repositories, pp.137–143, Shanghai, China, May 2006.
[178] D. Pagano and W. Maalej, "How do developers blog?: An exploratory study," Proc. Int'l Working Conf. on Mining Software Repositories, pp.123–132, Honolulu, HI, May 2011.
[179] FLOSSMetrics Final Reports, http://flossmetrics.org/ accessed Nov. 25, 2011.
[180] FLOSSmole, http://flossmole.org/ accessed Nov. 25, 2011.
[181] I. Herraiz, D. Izquierdo-Cortazar, F. Rivas-Hernandez, J. Gonzalez-Barahonam, G. Robles, S. Duenas-Dominguez, C. Garcia-Campos, J.F. Gato, and L. Tovar, "FLOSSMetrics: Free/libre/open source software metrics," Proc. European Conf. on Software Maintenance and Reengineering, pp.281–284, Kaiserslautern, Germany, March 2009.
[182] J. Howison, M. Conklin, and K. Crowston, "FLOSSmole: A collaborative repository for FLOSS reasarch and analyses," Int'l Journal of Info. Technology and Web Eng., vol.1, no.3, pp.17–26, July 2006.
[183] J.M. Gonzalez-Barahona, "Repositories with public data about software developmemt," Int'l Journal of Open Source Software and Processes, vol.2, no.2, pp.1–13, April 2010.
[184] percentage, http://zorba.knu.ac.kr/research/MSR_Survey/MSR09_11.html accessed Dec. 6, 2011.
[185] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," Proc. Int'l Working Conf. on Mining Software Repositories, pp.118–121, Cape Town, South Africa, May 2010.
[186] SourceForge, http://www.sourceforge.net/ Nov. 25, 2011.
[187] GoogleCode, http://code.google.com/ Nov. 25, 2011.

Woosung Jung received his B.S. and Ph.D. degree in Computer Science and Engineering from Seoul National University, Korea, in 2003 and 2011, respectively. He was a researcher in SK UBCare from 1998 to 2002. He was a senior research engineer at Software Capability Development Center in LG Electronics from Aug. 2011 to Feb. 2012. He is currently a full time lecturer at the Dept. of Computer Engineering, Chungbuk National University. His research interests include software evolution, software architecture, adaptive software system and mining software repositories.

Eunjoo Lee received her B.S., M.S., and Ph.D. degrees in Computer Science from Seoul National University, Korea in 1997, 1999, and 2005, respectively. She was a research staff member at Samsung Advanced Institute of Technology from Nov. 2005 to Feb. 2006. Currently, she is an assistant professor at the School of Computer Science and Engineering at Kyungpook National University. Her current interests include software reengineering, software metrics, web engineering, and mining software repositories.

Chisu Wu received his B.E. degree in applied mathematics from Seoul National University, Korea in 1972 and his M.S. and Ph.D. degrees in Computer Science from Seoul National University in 1977 and 1982, respectively. He served as a researcher at the Loughborough University, UK in 1978. From 1975 to 1982, he was an associate professor of Computer Science at Ulsan University, Korea. Currently, he is a professor of Computer Science and Engineering at Seoul National University, Korea. His current research interests include software engineering and programming languages.