Copyright ⓒ2017 KSII

Extracting the Source Code Context to Predict Import Changes using GPES

Jaekwon Lee[1], Kisub Kim[1], Yong-Hyeon Lee[2], Jang-Eui Hong[3], Young-Hoon Seo[1], Byung-Do Yang[4] and Woosung Jung[5]

[1] Department of Computer Engineering, Chungbuk National University,
[3] Department of Computer Science, Chungbuk National University,
[4] Department of Electronics Engineering, Chungbuk National University,
Cheong-ju, South Korea
[e-mail: {exatoa, falcon, jehong, yhseo, bdyang}@cbnu.ac.kr]
[2] Neowiz Games, Seongnam, South Korea
[email : gd9live@neowiz.com]
[5] Graduate School of Education, Seoul National University of Education,
Seoul, South Korea
[e-mail: wsjung@snue.ac.kr]
*Corresponding author: Woosung Jung

Received September 11, 2016; revised February 5, 2016; accepted February 27, 2017; published February 28, 2017

Abstract

One of the difficulties developers encounter in maintaining tasks of a large-scale software system is the updating of suitable libraries on time. Developers tend to miss or make mistakes when searching for and choosing libraries during the development process, or there may not be a stable library for the developers to use. We present a novel approach for helping developers modify software easily and on time and avoid software failures. Using a tool previously built by us called GPES, we collected information of projects, such as abstract syntax trees, tokens, software metrics, relations, and evolutions, for our experiments. We analyzed the contexts of source codes in existing projects to predict changes automatically and to recommend suitable libraries for the projects. The collected data show that researchers can reduce the overall cost of data analysis by transforming the extracted data into the required input formats with a simple query-based implementation. Also, we manually evaluated how the extracted contexts are similar to the description and we found that a sufficient number of the words in the contexts is similar and it might help developers grasp the domain of the source codes easily.

Keywords: source code change, context analysis, library recommendation, software repository, extracting model

A preliminary version of this paper was presented at APIC-IST 2016, and was awarded as an outstanding paper. This version additionally includes a extracting source code context to predict import changes. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2015R1C1A1A01054994, No. 2014M3C4A7030505).

https://doi.org/10.3837/tiis.2017.02.035 **ISSN : 1976-7277**

1. Introduction

Software maintenance accounts for most of the life cycle of software, and it is an expensive and time-consuming task [1]. If the maintenance of software is not done on a timely basis, it becomes impossible to prevent serious failures, and users will mistrust the software as well. However, predicting source code changes and recommending those changes to developers would resolve these problems and even improve the productivity and reliability of software projects. A recent study of the recommendation of source codes, libraries and components supports increasing the efficiency of development by mining data from version control systems (VCS) such as Git [2] and SVN [3]. Work by Ying helped to find suitable code snippets to meet the needs of developers by mining the patterns of source codes that frequently change together from each revision in VCS [4], and Zhong suggested a mining approach which helps developers find libraries easily by mining API usages [5]. Furthermore, a recommendation system for code snippets which collaborates the results of several code search engines and a heuristic algorithm was presented by Inoue [6]. Systems responding to user queries as in the approaches proposed above improved the precision of the system that recommends appropriate libraries. However, developers do not feel it necessary to search for new version of libraries unless an event occurs in an existing project. When they realize the necessity to update and search for certain libraries, a loss of information can occur often during the process of translating the images in mind into keywords for a query [7]. Moreover, because libraries which did not gain recognition by search engine algorithms [8] exist, it may be difficult to investigate the specific libraries desired by developers. For example, developers utilize a library, ListView, with numerous customizations for faster scrolling in the early stages of an Android project. RecyclerView, which is a container added to support various features of a previous version, retains a limited number of views to provide effective scrolling, meaning developers no longer have to customize ListView [9]. Similarly, the calendar class of JDK1.1 reduces the amount of realization for the partial inheritance extension class of JDK1.0.1 [10]. Developers who do not know these facts may use ListView instead of RecyclerView and use a Date class instead of a Calendar class, respectively, and thus incur a higher cost.

In this paper, we propose an overall approach for resolving the problems developers encounter in maintaining tasks of large-scale software systems by the updating of suitable libraries on time: predicting and recommending suitable libraries so that source codes can be imported. However, because of the need for accuracy in the extraction, we focused on analyzing the context of source codes in projects only to predict changes in codes automatically. The method we propose assumes that the developers do not need the library at the given time, that they missed it upon searching, or that they were not able to consider security and reliability. We explore changes in libraries imported into each source code with the evolution of the project and map the contexts between the points of changes in the libraries. By clustering and integrating this information, we extract the context of change imports for a specific library. We propose a tool called the General Purpose Extractor for Source Code (GPES) that extracts information about the structure, evolution, and quality from the Git repository to help researchers undertake source code analyses in various studies [11]. We collected data from twelve projects that were extracted by the GPES from GitHub and used these data to extract the contexts for revising import information. For the experiments, we obtained 568 revision sets from 5,909 revision in the projects.

The main contributions of the proposed approach are as follows:

- The GPES is able to support researches related to the evolution processes of software on various perspectives and levels by providing basic data such as structure, change, and quality information of the source codes that extracted from software repositories.
- Since GPES automatically extracts appropriate software information for the proposed schema, it can process necessary data for analysis in various forms with low cost queries.
- The proposed approach can help to predict future source code changes by analyzing the change pattern of the source code that is dependent on the import statements.

In this paper, we initially explore related studies in Section 2. In Section 3, the detailed processes used during the proposed approach are explained. Section 4 demonstrates our previously built tool. In section 5, experiments and case studies are presented. Lastly, we conclude our discussion with recommendations for future work in section 6.

2. Related Work

2.1 Context analysis and extraction

Studies that determine topics and domains with the contexts from source codes and that automatically generate summaries and software documents through a source code analysis have allowed made researchers to be able to investigate software from the perspective of evolution by comparing the contexts of each revision.

Maskeri et al. proposed a human-assisted approach [12] based on the investigated latent Dirichlet allocation (LDA) scheme for extracting domain topics from source code. LDA is a statistical method that has emerged as a popular technique for discovering domains and topics. Their method was applied to a number of open source systems, with the preliminary results indicating that LDA can identify several domains and topics. Furthermore, they suggested the method as a starting point for the additional manual refinement of topics.

With regard to better software maintenance and reuse methodologies, Hill et al. presented an approach [13] that extracts natural language phrases automatically from existing source code identifiers and categorizes the phrases into results in a hierarchy. Using this method, developers can explore word usage in pieces of actual code, identify relevant program elements to investigate, and recognize replaceable words for query reformulations. The empirical results with 22 developers showed that their method significantly outperforms others.

Using identifier names and comments from source codes to find topics, domains, and objectives was proposed by Kuhn et al. in 2007 [14]. They also applied the names of classes and packages to the computation of similarities using LSI to support the quality of the results.

Programmers use documents to understand software codes. However, according to an approach proposed by McBurney et al. [15], most software documents are written by humans in what is a time-consuming task as well. Therefore, McBurney et al. proposed a source code summarization technique that involves the writing of a description of each method of source code in Java after an analysis of the invocations. They compared the method to summaries written by humans and to a state-of-the-art method, demonstrating improvements over the state-of-the-art method in several dimensions.

2.2 Source code changes

In most software projects, a number of revisions are generated from creation to elimination. Developers update the versions of the source codes for various purposes, such as fixing bugs,

improving the speed or quality, and reducing the complexity. Research about source code changes is important in the software field, and numerous researchers work in this area.

We presented an approach to extract a developer's share of code in 2015. In that paper, we focused on diff values discovered by analyzing each code change. We broke down the source codes of each revision to abstract syntax tree to calculate the developer's share of codes and to analyze the systems effectively at the source code level [16].

Tao et al. [17] explained how software engineers understand code changes with actual examples, a number of surveys and common development scenarios. They found that the determination of the change risk is very important for understanding the changes and crucial needs for assessing the quality of a change, also finding that the quality of the description can affect the efforts to understand changes later. The paper ends with several suggestions indicating that engineers should understand the importance of change descriptions and should be responsible for providing more information.

In a paper written by Nguyen et al. [18], repetitiveness was defined as the ratio of repetitive changes over all changes, modelling a change as a pair of old and new ASTs within a method. They reported three findings. The repetitiveness of changes could increase by 70-100% in minor cases, and it decreases as the size increases, becoming higher and more stable in a cross-project setting, with the fixing of changes repeating similarly to general changes. The results of their learning and recommending system returned a value of about 30%, indicating that repeated the fixing of changes could be quite useful as a means of automatic software repair as well.

Discovering and identifying unknown change patterns were the purposes of Negara's study [19]. They applied a fine-grained sequence of code changes and found that their algorithm could handle challenges that distinguish continuous code change patterns using data mining techniques. In the evaluation phase, 1,520 hours of code development were collected from 23 developers, with the main result showing that the method was sufficiently effective and that it scales to large amounts of data.

2.3 Library searches and recommendations

Research on retrieving and recommending libraries and components to improve the productivity and quality of projects has been ongoing for more than a decade.

An approach proposed by Thung was a recommendation system for various APIs. The system is divided into two parts; one uses association rule mining and the other uses collaborative filtering. Each part of the system recommends a list of libraries and the aggregator then integrates the lists by means of heuristic weighting. To evaluate the approach, the researchers conducted ten-fold validation and consequently obtained a recall rate of approximately 80% [20].

In research for better reuse outcomes, Kalia et al. split the external factors of function, technology, integration framework, interoperability, portability, and non-functional characteristics, as well as the internal factors of encapsulation, the component type, architectural aspects, and accessibility to the source code. They then explained the improvement by the approach in terms of searching, understanding the components and in terms of detailed specifications [21].

Understanding suitable libraries or components by specific developers was also mentioned by Aziz et al. They tagged queries to names of classes and methods and then used a clone detection technique to construct patterns for code features. Subsequently, the system searched for components from libraries based on the clone patterns and finally applied a program slicing technique to help developers understand the retrieved components [22].

3. Extracting Change Context

In this paper, we propose a method that extracts contexts for recommending potential libraries to a currently existing project. The architecture of the method is shown in **Fig. 1**. If an input R_x which is a revision of a current project enters into the system, the system extracts from *Rx* to R_{x-t}, which is a proximate revision, as a RS (revision set) from the source code repository. After we extract all of the revisions of the RS, the system manufactures them to generate the *Change Context*. We treat a context from the first revision as a *Base Context* and compute the differences in the contexts from the second revision to the final revision, after which we integrate each of them to generate a *Change Context*.

The following figure explains the details of the extraction and the final purpose of the study.

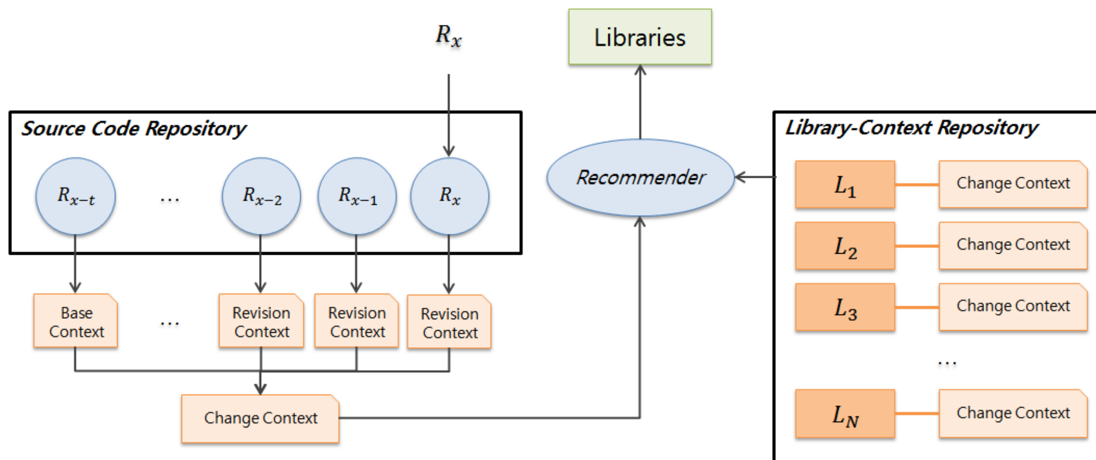


Fig. 1. The overview of library recommendation system with code context changes

3.1 Generating a Change Context

One RS has context information about a change that occurred up to the point of import. The *Change Context* contains the entire context of one RS and is constructed by integrating the contexts of the source codes from each revision.

We construct the context of each revision of the source code using word tokens extracted from the class, method, and variable names. The names represented in the source code represent their roles. There are two types of conventions: the camel case and the snake case. Developers usually apply these naming conventions to their codes, and occasionally they do not follow the conventions, such as *convertUTFtoASCII*, *NEGATIVEDECIMALTYPE*, and *actionparameters*. To solve this mixed-case splitting problem, Enslen et al. suggested Samurai [23], which is a tool that separates names of classes, methods, and variables to tokens based on mining approach. Samurai uses program-specific and global frequency tables to determine the split point of a mixed-case name extracted from 9,000 open-source projects. We apply this form of tokenizing and then remove the stopwords. Stopwords are those words that are very common in most documents such as a, an, the, is, at, and which, among others.

To combine the contexts of the revisions, we define the change context structure. The *Change Context* consists of three parts, a *Base, an ADDED* context, and a *DELETED* context. A *Base* context is a list of tokens with its count extracted from the first revision of the RS. To create an *ADDED* and *DELETED* context, we calculate the context differences between two

revision sequences. The context differences have ADDED and DELETED tokens. We merged all context differences into one *Change Context*. When merging all contexts, we consider the count. **Fig. 2** shows an example of the structure of a *Change Context*. The *Base* context has three *file* tokens and the ADDED context has one *file* token, indicating that the final state of this file has four *file* tokens.

```
{
  "Base":{"List":5, "load":1, "file":3 …},
  "ADDED":{"save":1, "convert":1, "file":1},
  "DELETED":{}
}
```

Fig. 2. An example of a Change Context

3.2 Build Library Repository

To recommend libraries, the mapping information of libraries related to the changed context is necessary. In this subsection, we report a means of mapping between the extracted RS from an existing project repository and a library.

The RS is defined as a set of revisions between one revision that has an import change in a certain source code and the other revision that has the next import change. **Fig. 3** shows that the revisions in a source code repository. When the revisions were changed from R_1 to R_6 in a regular sequence, and if there were import changes R_3 and R_6, $RS1 = \{R_1, R_2\}$, $RS2 = \{R_3, R_4, R_5\}$, at this point of the phase, we need information about the library and mapping RSs from the repository to recommend libraries. Because the information pertaining to the library exists on R_3 and R_6, which have actual import changes, the system should expand and include R_3 and R_6 in *RS1* and *RS2*, respectively. For the last revisions R_3 and R_6, we extract the changed import information and map to the context.

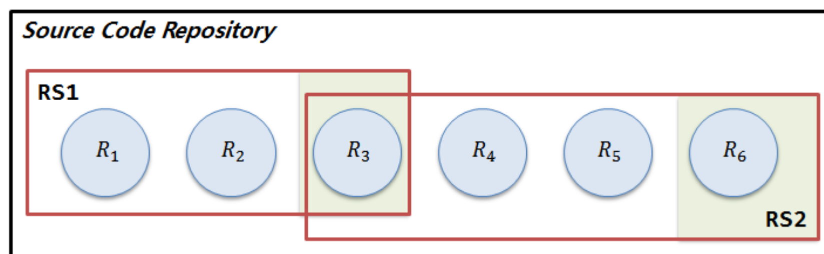


Fig. 3. An example of a revision set in a source code repository

Some of the extracted RSs are not useful. Regarding the changes in each revision, there is information related to a library change as well as unrelated information. Specifically, when multiple changes occur at once, it is difficult to classify them if there is any relationship between the libraries and the contexts. Therefore, we exclusively consider RSs that have only one library change for recommendations.

During the development process, developers make multiple branches, and the revision history has a graph of all the branches. **Algorithm 1** shows the pseudo code used for extracting the RSs of a specific file, given *FilePath* as a parameter, using a depth-first search (DFS). The algorithm gets a graph information via calling a *getAllRevisionsChildren* module and generates the RSs through a module termed *SearchRS*, which explore the nodes recursively. The module requires two parameters. The first parameter is the *StartNode*, which is the first

revision of the file; and the last one is *RSPath*, which is a sequence of nodes that will be included in the current RS. The module *SearchRS* calculates the differences in the import information between all the node pairs in the current node. If there are no differences, it moves to a child node while retaining the status of the parameters. If this is not the case, it moves to the child node after initializing *RSPath.* In the condition set up here, only one different import change occurs, and the modules build a RS and retain it. If the system goes back and forth during the process above until visiting all the leaf nodes are visited, it generates every RSs for the file.

To determine the import difference between neighboring nodes, the system obtains an import list of all nodes and compares them. This can be done during the DFS exploration step, but the performance must be considered. Thus, making the calculation after obtaining the import differences between neighboring nodes via a module termed *makeAllRevisionsDiff* is a better method. The module utilizes a breadth-first search (BFS) approach to search for neighboring nodes.

Algorithm 1. makeRevisionSet(FilePath, StartNode)

```
Diffs = makeAllRevisionsDiff(FilePath, StartNode)
ChildrenList = getAllRevisionsChildren(FilePath)
RevisionSets = list()
visited = dictionary()
SearchRS(StartNode, list())
return RevisionSets

function SearchRS(node, RSPath):
    if node in visited then return
    RSPath += node
    visited[node] = True
    children = ChildrenList[node]
    if length(children) == 0 then return

    for child in children do
        diff = Diffs[node][child]

        if isNoChanged(diff) is True then
            SearchRS(child, RSPath)
        else
            if isOneChanged(diff)then
                RevisionSets += RS(RSPath+child, diff)
            end if
            SearchRS(child, list())
        end if
    end for
end function
```

4. GPES (General Purpose Extractor for Source Code)

In this section, to support a general purpose analysis of source codes, we previously proposed a tool known as GPES to extract raw data such as the version, abstract syntax tree (AST), differences, and metrics of the source code from the Git repository and then to store the data in

a predefined schema. GPES provides flexible customization of the data source for MSR researchers based on the extraction tool and the schema. Our tool can be differentiated from the state-of-the-art tool BOA [24] in terms of how it supports revision history information, revision snapshots, and various software metrics for multiple levels of source code.

4.1 Extraction Process

The overall process of GPES consists of three phases, as shown in **Fig. 4**.

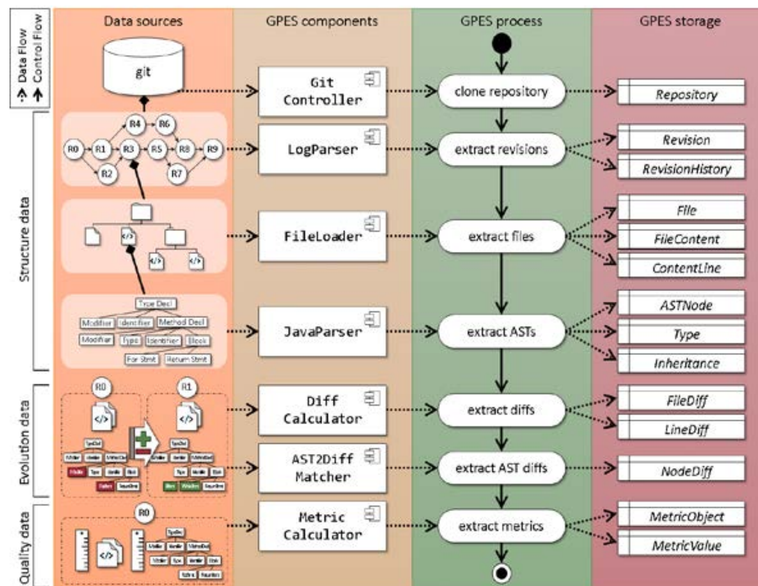


Fig. 4. Overall process of GPES

The first phase is the process of extracting the structure data, such as revisions, files, and ASTs, from the source codes. By analyzing the logs from a cloned remote repository, this phase starts with the storing of metadata and history graphs of revisions, after which it saves the metadata and textual data of the files extracted from each revision. Our tool generates ASTs from the source codes using `Eclipse JDT` and then stores all AST nodes with the data types and their inheritance information.

In the next phase, our tool extracts evolution data that corresponds to the changesets of the files, the code lines, and the nodes of the ASTs by comparing the two revisions. The changesets of the files and lines can be gained by parsing the line-level-diff results, whereas the changeset for AST is extracted through a comparison between each node of the AST and the results calculated by word-level-diff.

Finally, using an external static analysis utility called `Understand`, our tool extracts the quality data measured by various software metrics for objects such as files, packages, classes, and methods. If the measured object is an overloaded method or an anonymous class, the objects are mapped in order of appearance in the metric results because the qualified name conflicts with others.

4.2 Database Schema

The database schema of GPES has three models for the structure, evolution, and quality associated with each extraction phase. **Fig. 5** shows tables of each model and the relationships

among them within our schema. The structure model includes tables for revisions, developers, files, and ASTs. We store the metadata of the versions in *Revision* and the ordering relationship between two revisions in *RevisionHistory*. Because only a few files are changed in each revision, all files of each version are saved as *File* and *FileContent* by separating them into the metadata of files and their own content, respectively. *ASTNode* contains node elements such as declaration, expression, statement, and the identifier of the AST. *Type* and *Inheritance*, on the other hand, correspondingly include primitive and object-oriented data types and their inherited relationships. The evolution model consists of the three tables of *FileDiff*, *LineDiff*, and *NodeDiff*, which are matched with differences between files, code lines, and AST nodes. The quality model has two tables; one is *MetricObject*, which links objects corresponding to packages, files, classes, and methods to be measured by metrics; the other is *MetricValue*, which contains various metric counts for each object.

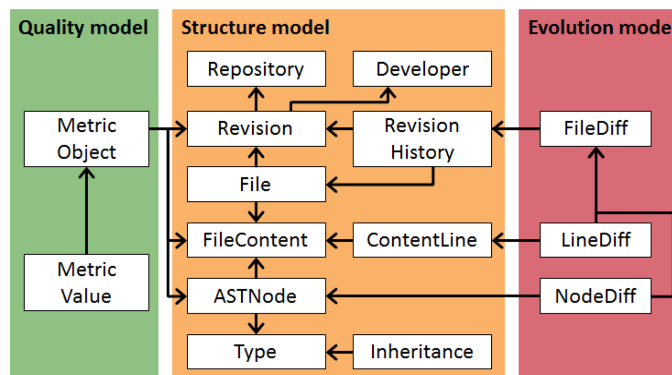


Fig. 5. Database schema

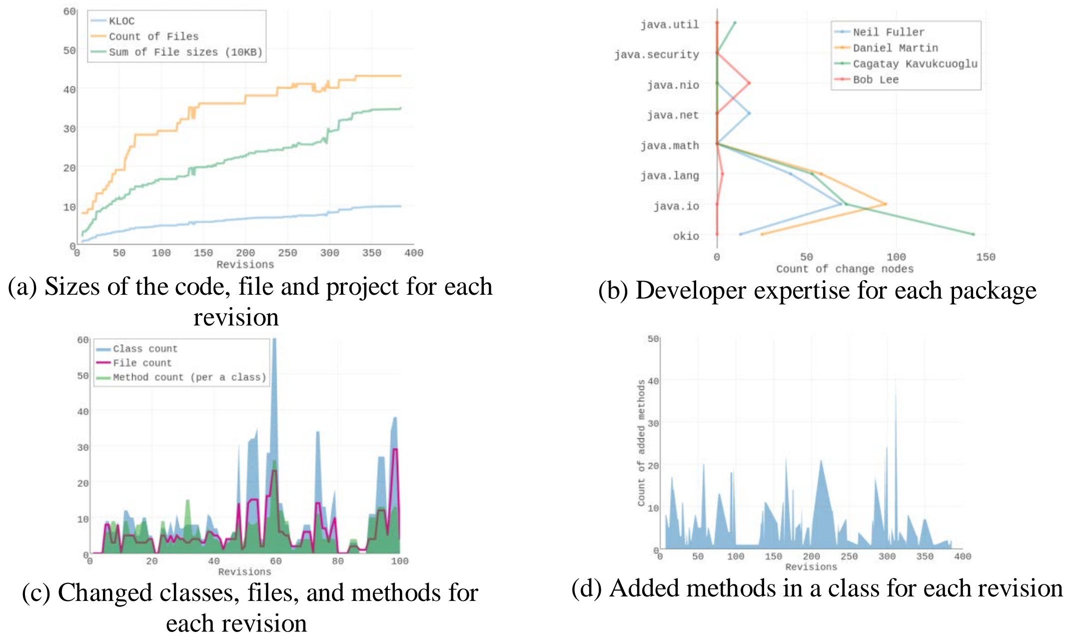
5. Experiments

We conducted case studies to evaluate what types of research areas GPES can cover and how much the cost can be reduced in the phases for the previous study and the accuracy of recommendation for the currently proposed approach. We chose the Okio project, which is based on the Java language and which satisfies the conditions of the upper five releases, five contributors, and 1000 stars among the projects, as the target.

5.1 Input Data Processing

To validate the possibility of supporting various research areas, we extracted evolution metrics [25], developer expertise [20], change coupling [26], and source code differencing [27] from previous studies using our tool, GPES. **Fig. 6** shows visualizations of parts of the data from each type of research.

(a) shows the change histories of the lines of code (LOC), the number of files, and the sum of the file sizes as revision changes. (b) represents the number of AST nodes in each package that each developer changed. (c) shows the number of changed files, classes, and methods per class. The number of added methods in existing classes for each revision is signified in (d). Based on these visualizations, we can analyze the patterns or trends in the data, such as developer expertise. Thus, the GPES schema can provide plenty of data to study the MSR through processing with queries.



(a) Sizes of the code, file and project for each revision

(b) Developer expertise for each package

(c) Changed classes, files, and methods for each revision

(d) Added methods in a class for each revision

Fig. 6. Visualization results for input data for each part of the research on the Okio project

5.2 The Cost of Data Processing

As a means of demonstrating the effectiveness of our tool, we estimated the cost of preprocessing by queries to obtain the data with which to study the MSR using GPES. **Table 1** shows the statistics for the SELECT queries, the tables used, and other queries (e.g., CREATE, INSERT, UPDATE, CURSOR). We used five tables, two to three SELECT queries, and six other queries on average. Therefore, it was found to be more effective to deal with data using GPES than to process the data directly from a repository.

Table 1. Usage Counts for Query and Table for Each Research Subject

Subject	SELECT **queries**	**Used tables**	**Other queries**
Evolution metrics [25]	5	3	4
Developer Expertise [20]	2	6	10
Change Coupling [26]	1	4	10
Source code differencing [27]	1	5	0

5.3 Performance of the extraction

The time complexity of the algorithm that we proposed to analyze a code revision graph is $O(|E|)$, where E is a set of edges that is represented the relationships between revisions. The performance of **Algorithm 1** depends on two modules, *makeAllRevisionsDiffs* and *SearchRS*, which are implemented based on BFS and DFS techniques, respectively. These techniques have a time complexity of $O(|V| + |E|)$ for vertex set V and edge set E when the algorithm retrieves all vertices as a starting point. However, because our algorithm only needs to retrieve a graph from a first revision vertex, the performance of the algorithm depends on the number

of edges. Fig. 7 shows a graph of the execution time for the algorithm that increases linearly as the number of edges increases. This means that we can extract RSs in finite time if we do not analyze files with enough revisions to cause a stack overflow.

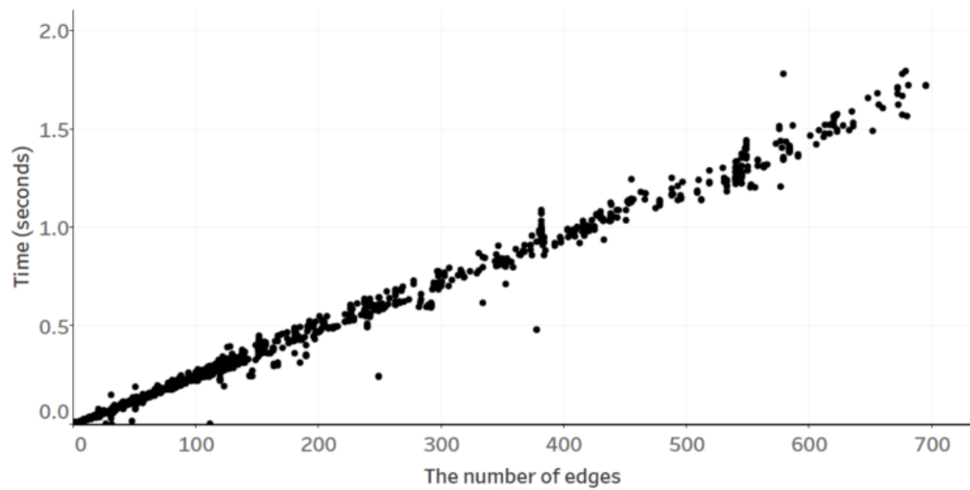


Fig. 7. The performance of the RS extracting algorithm

5.4 The Extraction of Context

We manually evaluated the part for extracting contexts from source codes based on import changes.

First, we used Samurai tool that enables researchers to extract tokens. For using the Samurai, we need global frequency table that uses tokens from a large number of projects. Because GPES had a limitation on the speed, we partially used Boa Infrastructure. Consequently, we collected 7,988 open source projects in GitHub while Enslen collected 9,000 open source projects.

For evaluating the performance of extraction for change context, we extracted 10,420 RSs from 12 projects extracted by GPES. We filtered out the noise from RSs with no tokens and RSs only contains the BASE contexts as the pre-processing step. Manual evaluation was conducted with 586 RSs and 211 libraries. The experiment was confirming the existence of intersections between contexts from library descriptions and change contexts.

Table 2 shows the results for a simple output and a comparison. We considered all 211 libraries manually and discovered that which libraries may be imported into specific revisions through analyzing the point of changing contexts. Furthermore, if the number of intersections actual description is higher than three, the context we extracted is indicated the domains and topics of the libraries.

5.5 Threats to validity

We had the following threats to validity in our experiments.

Since our approach only utilizes code change histories, inferring the immediate cause for the code change or recommending another version of the same library are difficult. For example, suppose we have two same libraries with different version and one resolved a vulnerability and the other did not. If we extract the context from the codes that used the two libraries, we cannot recognize the difference between the two libraries. Likewise, the RS context does not fully reflect the inside of a particular library. This limitation can be resolved

by appending information such as analyzed commit messages or descriptions for the libraries. However, a commit message accumulated along with a revision set can cause confusion in the RS context for the library. We therefore, limited this study to extracting the context from the statistical data obtained from the GPES-based source code.

The experiment was conducted based on the data extracted from the GPES; which was implemented for Java languages although it designed for the general purpose. This can result in being biased for a particular language. However, since our approach creates contexts based on information available from code identifiers, it is more influenced by the naming conventions of identifiers than by grammatical features. For this reason, experiments on other languages are less important. Therefore, we have experimented only with the java language and have focused on using GPES data for general purpose use. If users want to experiment with other languages, they can add a parser for the language to the GPES to expand.

Table 2. The result for comparing tokens between change context and library description

Library Name	The number of tokens intersection with description	The count of change context tokens	The percentage of intersection
java.lang.annotation.Annotation	10	1	10%
java.nio.MappedByteBuffer	29	3	10%
java.util.logging.Logger	21	3	14%
java.io.Closeable	10	1	10%
com.corundumstudio.socketio.Configuration	22	6	27%
com.corundumstudio.socketio.parser.Packet	11	3	27%
com.corundumstudio.socketio.ack.AckManager	16	4	25%
java.util.regex.Pattern	76	11	14%
org.jboss.netty.channel.Channel	27	3	11%
com.corundumstudio.socketio.handler.PacketHandler	30	4	13%
java.util.zip.Deflater	22	5	23%
java.io.Flushable	10	3	30%
java.util.Random	203	27	13%

6. Conclusion

We proposed an overall approach for helping developers modify software easily and on time to prevent software failures. We focused on analyzing the contexts of source codes in existing projects to predict changes in the code automatically for this time. The method reported here assumed several situations of problems during the development process. To resolve the problems in the referenced situations, we extract the context of the change import for a specific library by clustering and integrating the mapping information. We evaluated our approach using our previously built tool GPES, with data collected from twelve projects that were extracted by GPES from GitHub and then used these data to extract the contexts to predict the import information. Our tool, GPES, offers several advantages. First, because GPES includes

a relationship graph for each revision, it can be used to analyze a project with different information between each revision. Second, it is advantageous to extract snapshot information because it includes metadata for each revision relative to the current state-of-the-art method. For the proposed approach, limitations such as including unintended and unrelated features in the imported libraries when the system extracts the context of the revision set can arise. We can resolve such limitations by integrating a bug tracking system to filter out unrelated context. Because the system considers revision sets that have only one import change, a limitation exists regarding the variety of environments as well. We will consider eliminating the limitations and recommending the suitable libraries to the projects as our future work. Furthermore, Expanding to revision sets that have more than one import change will support developers in various environments as well.

References

[1] K. Bennett and V. Rajlich, "Software Maintenance and Evolution: a Roadmap," in *Proc. of the Conference on the Future of Software Engineering*, pp. 73-87, June 4-11, 2000. Article(CrossRef Link).
[2] Git – fast version control system. Available from: http://git-scm.com
[3] Apache Subversion. Available from: http://subversion.apache.org
[4] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574-586, September, 2004. Article (CrossRef Link).
[5] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *Proc. of the 23rd European Conference on ECOOP*, pp. 318-343, July 6-10, 2009. Article (CrossRef Link).
[6] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where Does This Code Come from and Where Does It Go? - Integrated Code History Tracker for Open Source Systems -," in *Proc. of the 34th Int. Conference on Software Engineering*, pp. 331–341, June 2-9, 2012. Article (CrossRef Link).
[7] D. Lucrédio, A. F. Do Prado, and E. S. De Almeida, "A survey on software components search and retrieval," in *Proc. of the 30th Euromicro Conference*, pp. 152-159, September 3-3, 2004. Article (CrossRef Link).
[8] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," *World Wide Web Internet Web Information System*, vol. 54, no. 2, pp. 1–17, January 29, 1998. Article (CrossRef Link).
[9] Creating Lists and Cards, Available from: https://developer.android.com/training/material/lists-cards.html
[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1st Edition, Addison-Wesley, 1999.
[11] J. Lee, Y. Lee, K. Kim, J. Hong, and W. Jung, "GPES : Supporting Source Code Analysis by Extracting the Evolutionary History of Software Structure and Quality," in *Proc. of the 11th Asia Pacific Int. Conference on Information Science and Technology*, pp. 43-45, June 26-29, 2016. Article (CrossRef Link).
[12] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proc. of the 1st India Software Engineering Conference*, pp. 113-120, February 19-22, 2008. Article (CrossRef Link).
[13] E. Hill, L. Pollock, and K. V. Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *Proc. of the 31st Int. Conference on Software Engineering*, pp. 232-242, May 16-24, 2009. Article (CrossRef Link).
[14] A. Kuhn, S. Ducasseb, and T. Gîrbaa, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230-243, March 2007. Article (CrossRef Link).

[15] P. W. McBurney, and C. McMillan, "Automatic Source Code Summarization of Context for Java Methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 02, pp. 103-119, February, 2016. Article (CrossRef Link).

[16] Y. Lee, K. Kim, and Woosung Jung, “Analyzing Developer’s Share of Code Based on Abstract Syntax Tree,” in *Proc. of the Korean Society of Computer Information Conference*, Vol. 23, No. 2, pp. 23-24, July 9-11, 2015. Article (CrossRef Link).

[17] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim "How do software engineers understand code changes?: an exploratory study in industry," in *Proc. of the 20th Int. Symposium on the Foundations of Software Engineering*, pp. 51:1-51:11 November 11-16, 2012. Article (CrossRef Link).

[18] H.A. Nguyen, A.T. Nguyen, T.T. Nguyen, T.N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proc. of the 28th Int. Conference on Automated Software Engineering*, pp. 180-190, November 11-15, 2013. Article (CrossRef Link).

[19] S. Negara, M. Codoban, D. Dig, and R.E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proc. of the 36th Int. Conference on Software Engineering*, pp. 803-813, May 31-June 7, 2014. Article (CrossRef Link).

[20] F. Thung, D. Lo, and J. Lawall, “Automated library recommendation,” in *Proc. of the 20th Working Conference on Reverse Engineering*, pp. 182–191, October 14-17, 2013. Article (CrossRef Link).

[21] A. Kalia and S. Sood, “Characterization of Reusable Software Components for Better Reuse,” *Int. Journal of Research in Engineering and Technology*, Vol. 03, No. 05, May 2014. Article (CrossRef Link).

[22] M. Aziz and S. North, “Retrieving software component using clone detection and program slicing,” Sheffield, UK: The University of Sheffield, February 2007. Article (CrossRef Link).

[23] E. Enslen, E. Hill, and L. Pollock, “Mining Source Code to Automatically Split Identifiers for Software Analysis,” in *Proc. of the 6th Int. Working Conference on Mining Software Repositories*, pp. 71–80, May 16-17, 2009. Article (CrossRef Link).

[24] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A Language and Infra- structure for Analyzing Ultra-Large-Scale Software Repositories,” in *Proc. of the 35th Int. Conference on Software Engineering*, pp. 422-431, May 18-26, 2013. Article (CrossRef Link).

[25] A. Capiluppi, M. Morisio, and J. F. Ramil, “Structural evolution of an open source system: a case study,” in *Proc. of the 12th Int. Workshop on Program Comprehension*, pp. 172-182, June 26-26, 2004. Article (CrossRef Link).

[26] T. Zimmermann, A. Zeller, P. Weißgerber, S. Diehl, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429-445, June, 2005. Article (CrossRef Link).

[27] J. I. Maletic, M. L. Collard, “Supporting source code difference analysis,” in *Proc. of the 20th Int. Conference on Software Maintenance*, pp. 210–219, September 11-14, 2004. Article (CrossRef Link).

Jaekwon Lee received his B.S.E. and M.Eng. degrees in Computer Engineering from Chungbuk National University, Korea, in 2013 and 2015, respectively. Currently, he is a Ph.D Student in the Department of Computer Engineering, Chungbuk National University. His research interests include mining software repositories, software evolution and search-based software engineering.

Kisub Kim received his B.S.E. degree in Computer Engineering from Chungbuk National University, Korea, in 2014. He was a student researcher at Natural Language Processing Lab in Chungbuk National University from 2012 to 2014. He was a developer in Kyunghee University Medical Center from 2014 to 2015. He is currently an M.Eng. student at the Dept. of Computer Engineering, Chungbuk National University. His research interests include mining software repositories, code search, and software evolution.

Yong-hyeon Lee received his B.S.E. and M.Eng. degrees in Computer Engineering from Chungbuk National University, Korea, in 2014 and 2016, respectively. He is currently a member of Game Platform Development Team at NEOWIZ GAMES Corp. His research interests include software evolution and source code analysis.

Jang-Eui Hong is a professor of Computer Science Department at the school of Electrical and Computer Engineering, Chungbuk National University, Cheongju, Korea. He received his Ph.D in computer science from KAIST, Korea, in 2001. He served as a research member at ADD(Agency for Defense Development) from 2000 to 2002, and also served as a principal consultant at SolutionLink, Co., Ltd. His research interests include software quality, embedded software architecture, low-energy software model, and software process improvement.

Young-Hoon Seo received his B.S., M.S., and Ph.D degrees in Computer Engineering from Seoul National University, Korea, in 1983, 1985, and 1991, respectively. He was a visiting scholar at the Center for Machine Translation, Carnegie-Mellon University from 1994 to 1995. He is currently a professor at the Dept. of Computer Engineering, Chungbuk National University from 1988. His research interests include Natural Language Processing, Korean Language Analysis, Word Sense Disambiguation, Information Retrieval, Question-Answering.

Byung-Do Yang received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer science from Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea, in 1999, 2001, and 2005, respectively. He was a senior engineer at the Memory Division, Samsung Electronics, Kyungki- Do, Republic of Korea, in 2005, where he was involved in the design of DRAM. In 2006, he joined the department of electronics engineering, Chungbuk National University, Republic of Korea, where he is currently a professor. His research interests include circuit and system designs.

Woosung Jung received his B.S. and Ph.D. degrees 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 2011 to 2012. He was an associate professor at the Dept. of Computer Engineering, Chungbuk National University from 2012 to 2016. He is currently an associate professor at the Graduate School of Education, Seoul National University of Education. His research interests include software education, software engineering, adaptive software system and mining software repositories.