2013 20th Asia-Pacific Software Engineering Conference

Seamless Code Reuse with Source Code Corpus

Tetsuo Yamamoto
College of Engineering, Nihon University
Koriyama, Fukushima, Japan
Email: tetsuo@cs.ce.nihon-u.ac.jp

Norihiro Yoshida
Nara Institute of Science and Technology
Ikoma, Nara, Japan
Email: yoshida@is.naist.jp

Yoshiki Higo
Osaka University
Suita, Osaka, Japan
Email: higo@ist.osaka-u.ac.jp

***Abstract*—Code reuse is attracting much attention as a promising technique for efficient software development. However, code reuse itself requires human resources: for example, searching and opening source files including code fragments that users would like to reuse, or considering keywords in using code search systems. The present paper proposes a novel technique that hardly requires such reuse cost. In the proposed technique, what programmers have to do for obtaining reusable code is just inputting a trigger key for code reuse on their development environments. Also, this paper describes some applications on OSS with a prototype tool working on Eclipse.**

I. Introduction

Code reuse is one of the promising approaches for realizing efficient software development. At present, there are various kinds of methodologies and software tools for helping code reuse. Copy-and-paste is the most popular and the easiest way for code reuse, which is refer to cloning [13]. By using copy-and-paste, we can implement required functionalities within a short timeframe. However, code reuse itself requires costs, which is more expensive than finding reused code. For example, we need to search code fragments to be available for code reuse.

For solving problems related to code reuse, keyword-based code search systems have been developed [2], [3], [4]. If programmers give keywords associating functionalities required for developing systems, then search systems return code related to the keywords. By using such systems, we can avoid searching reusable code by ourselves. Another advantage of the systems is that reuse target becomes huge. Code search systems can search code being open to the public. On the other hand, when a programmer uses copy-and-paste, reuse target is only the code fragment that he/she or his/her team implemented in the past.

Also, techniques related to collaborative filtering are used in the context of software reuse [11], [15]. In collaborative filtering approaches, firstly, sets of libraries, APIs, or components that were used together in the past are extracted from historical data or obtained by automatic project monitoring. Then, recommending systems suggest reusable software assets by matching the collected sets and the developing context. Developers do not need to consider keywords for searching reusable software assets in collaborative filtering approaches. However, these approaches only suggest reusable asset. They do not show how reusable assets should be reused.

In recent years, collective intelligence attracts much attention. There are various kinds of research and systems using collective intelligence (e.g, *google suggest*) [6]. These systems firstly collect and analyze data of a large number of people. Then, they presume what users want at present. It is highly possible that collective intelligence can be a great help for code reuse. Suggesting code based on collective intelligence database created from open source software should be helpful because programmers can obtain actual code implemented by skillful developers of open source software.

This paper proposes a novel technique for seamless code reuse by using hash-based code clone detection techniques. The features and contributions of this paper are as follows:

- In the proposed technique, queries for searching code are the half-written code. He/she does not need to consider keywords for searching code. All he/she has to do is triggering code search. Consequently, the time required for code reuse is much shorter than keyword-based code search systems.
- Reusable code is identified based on hash-based clone detection [5], [12]. Consequently, the proposed technique find reusable code even if its comments or identifier names are not exactly the same as the writing code.
- The proposed technique has been implemented as a Eclipse plugin. Nowadays, Eclipse is widely used, and so the implementation can be helpful for many programmers.

The remainder of this paper is organized as follows. Section II describes the proposed technique and its implementation. Section III shows experiments conducted with the implementation. Section IV introduces related works, and Section V concludes this paper.

II. Proposed Technique

Herein, we explain a new technique realizing seamless code reuse. Figure 1 shows an overview of the proposed technique. It consists of two processes. One is **corpus creation process**, which creates a code corpus from target source files. The created corpus is stored into a database. The other is **corpus usage process**, which suggests reusable code based on developers' reuse requests. When a programmer is implementing a functionality, code stored in the corpus is recommended on an integrated development environment that the programmer are using. The two processes are independent, both the processes function separately.

The remainder of this section explains both the processes and its implementation.

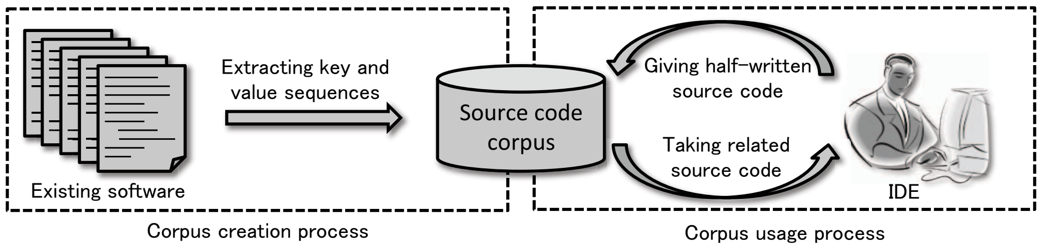

Extracting key and value sequences
Source code corpus
Existing software
Corpus creation process
Giving half-written source code
Taking related source code
IDE
Corpus usage process

Fig. 1. Overview of the proposed method

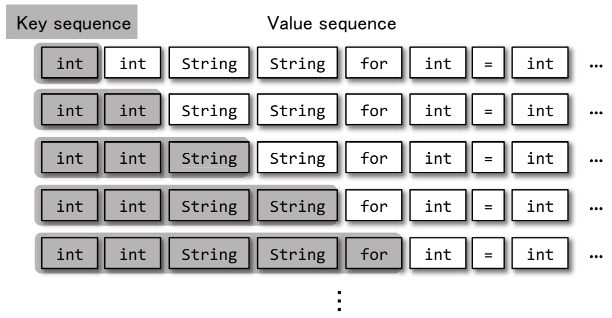


Fig. 2. Key and value sequences

A. Corpus Creation Process

The *corpus creation process* consists of **code analysis** and **appearance counting**.

- The *code analysis* extracts all the modules from given source code, and then transforms them into a special format to store in the corpus. The source code itself is not stored in the corpus.
- The *appearance counting* checks the number of appearances for each element in the corpus and reorders the elements in the order of the count.

In the *code analysis*, modules are extracted from the input source files. Then, every module is converted to token sequences. In this step, variable names are changed to their type names. Every token sequence is divided into two parts at every border of consecutive tokens. If there are n tokens in a token sequence, there are $n-1$ dividing points. Herein, we call the anterior part **key sequence** and the posterior part **value sequence**, respectively. Figure 2 shows a dividing example.

Next, generated pairs are registered into the corpus. There can be multiple different value sequences for the same key sequence. Also, different pairs can have both the same key sequence and the same value one. Consequently, we use a triplet of key sequence, value sequence, and appearance count for registering key and value sequences into the corpus.

When pairs of key and value sequences are being registered into the corpus, their appearances are counted as follows.

- **STEP1:** checking whether a given pair is the same as any pairs in corpus or not. Herein "the same pair" means both the key and value sequences are exactly the same.
- **STEP2A:** if there is not the same pair, a new triplet is created for the given pair. The appearance count of the triplet is 1. Then, it goes to STEP3.
- **STEP2B:** if exists, the appearance count of the triplet is incremented by 1. Then, it goes to STEP3.
- **STEP3:** triplets in the corpus are reordered by the appearance count.

The above steps are performed for every pair. The reordering (STEP3) is intended for efficiently providing necessary triplets in the corpus usage process.

B. Corpus Usage Process

The *corpus usage process* takes a half-written code on IDE and provides code related to it. The provided code is sorted in the order of its appearance count. In this process, the proposed technique monitors source code on IDE, and it converts the half-written code into a key sequence at the background. The conversion is the same as the *code analysis* in the *corpus create process*. Then, it queries the generated key sequence to the corpus and obtains triplets where the key sequence is included. If two or more triplets are returned, they are sorted in the order of their appearance count. The value strings of the obtained triplets are inversely converted to source code. The converted source code is presented to developers on IDE.

C. Implementation

We have implemented the proposed technique as a Eclipse plugin. Before the implementation, we had investigated Java open source software to derive some heuristics. As a result, we had found the following trend in Java source code.

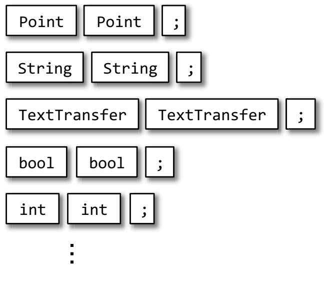
- Many methods include error checking code or assert code at the beginning of them (e.g., null checkings for avoiding NullPointerException).

With consideration for the trend, we decided to remove error checking code or assert code from source code for improving the accuracy of code completion. If we do not remove error checking code, developers need to write beginning of methods with error checking code. If there is not error checking code, reusable code is not suggested. The followings are the transformation rules that we adopted.

- All the tokens representing type names and variable names are converted into class names. All the comments and ".", which are used in method invocations are

```
public void paste() {
   bool flag;
  int count;
  TextTransfer transfer = TextTransfer.getInstance();
  Point selection = browser.combo.getSelection();
  String text = browser.combo.getText();
  int length=0;
  String newText= (String)browser.getContents(transfer);
      ⋮
```

(a) original source code



```
Point Point ;
String String ;
TextTransfer TextTransfer ;
bool bool ;
int int ;
⋮
```

(b) generated token sequence

Fig. 3. Example of token sequence generation on variable declarations

ignored. "{" and "}" are automatically inserted at the place where they are omitted.

- "if (expression==null) statement" is ignored. Herein, "expression" and "statement" mean any expression and any statement of Java, respectively.
- "if (expression1) return expression2;" is ignored. Herein, "expression1" and "expression2" mean any expressions of Java.
- "if (expression1) throw expression2;" is ignored. Herein, "expression1" and "expression2" mean any expressions of Java.
- Multiple variable declarations at the beginning of the method are merged as a single variable declaration. All variable initializers located in the right hand of variable declaration statements are ignored. Also, the variable declarations are sorted in the alphabetical order (case sensitive). Example of this process is shown in Figure 3. Source code shown in Figure 3(a) is the beginning of a method. Figure 3(b) shows a token sequence generated from the source code.

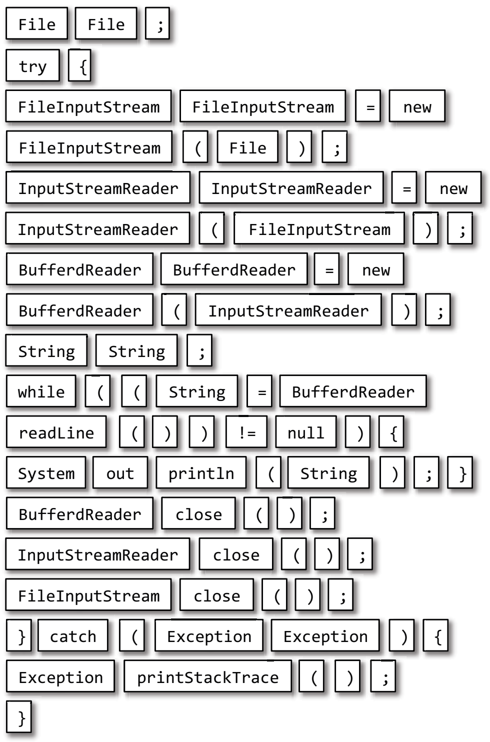
Herein, we explain the *code analysis* with an example of Java method in Figure 4. All the tokens in the method printFile are converted as shown in Figure 4(b). If a type name appears as a fully qualified name, simple name of the class is used. For example, a type is described as "java.io.File" in source code, the type is converted into "File". There is neither open nor close bracket ("{" and "}") for the while-statement in the source code meanwhile a pair of brackets exist in the token sequence.

Figure 5 shows a screenshot of the developed system. When a programmer is writing code on Eclipse, the plugin queries on the corpus by his/her trigger. The response of the system for the queries is extremely rapid because matching is performed based on hash values of key sequences. The matching results are presented as a list on Eclipse. If he/she selects a code fragment in the list, then it is pasted on the editor.

```
public class Sample1 {
  private int x;
  private int y;

  /**
   * Writes file contents to stdout
   */
  public void printFile(String filename) {
    if (filename == null) { return; }
    File file = new File(filename);
    try {
      FileInputStream is = new FileInputStream(file);
      InputStreamReader sr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(sr);
      String str;
      // Reads until end of file
      while ((msg = br.readLine()) != null)
        System.out.println(str);
      br.close();
      sr.close();
      is.close();
    } catch(Exception e) {
      e.printStackTrace();
    }}}
```

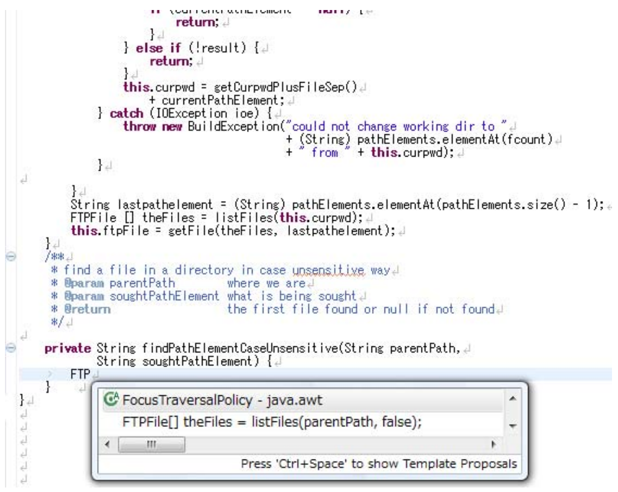
(a) original source code



```
File File ;
try {
FileInputStream FileInputStream = new
FileInputStream ( File ) ;
InputStreamReader InputStreamReader = new
InputStreamReader ( FileInputStream ) ;
BufferdReader BufferdReader = new
BufferdReader ( InputStreamReader ) ;
String String ;
while ( ( String = BufferdReader
readLine ( ) ) != null ) {
System out println ( String ) ; }
BufferdReader close ( ) ;
InputStreamReader close ( ) ;
FileInputStream close ( ) ;
} catch ( Exception Exception ) {
Exception printStackTrace ( ) ;
}
```

(b) generated token sequence

Fig. 4. Example of token sequence generation on Java method code



```
                    return;
                }
            } else if (!result) {
                return;
            }
            this.curpwd = getCurpwdPlusFileSep()
                + currentPathElement;
        } catch (IOException ioe) {
            throw new BuildException("could not change working dir to "
                                    + (String) pathElements.elementAt(fcount)
                                    + " from " + this.curpwd);
        }

    }
    String lastpathelement = (String) pathElements.elementAt(pathElements.size() - 1);
    FTPFile [] theFiles = listFiles(this.curpwd);
    this.ftpFile = getFile(theFiles, lastpathelement);
}
/**
 * find a file in a directory in case unsensitive way
 * @param parentPath          where we are
 * @param soughtPathElement   what is being sought
 * @return                    the first file found or null if not found
 */

private String findPathElementCaseUnsensitive(String parentPath,
        String soughtPathElement) {
    FTP
}
}
```

FocusTraversalPolicy - java.awt
FTPFile[] theFiles = listFiles(parentPath, false);
Press 'Ctrl+Space' to show Template Proposals

Fig. 5. A screenshot of the system

III. Experiment

A. Performance

Herein we describe an evaluation on the performance of the implemented plugin. We investigated timing of building database and size of the database. Table I shows a list of software that was used in this evaluation. Note that Apache Project is a set of software systems developed in Apache Project and they are open to the public in http://www.apache.org/.

Table II shows the investigation result. The timing of building database increases in proportion to the size of the target. Especially, in the case of Apache Project, it took about 13 hours to build a database. However, such a batch building is necessary only once at the beginning of usage. After the batch building, we can update the database for only updated files. Consequently, it is a matter of no importance that batch building requires much time.

Database size also increases in proportion to software size. In the case of Apache Project, the size is about 28GB, which is a reasonable size stored into storage of personal computers.

TABLE I
Target Software

Software	# of files	LOC
Ant 1.8.1	829	212,401
JDK 1.6.0	7,154	2,071,178
Apache Project	51,777	9,669,445

TABLE II
Time of Building Database and Database Size

Software	Building time	DB size
Ant 1.8.1	3 mins. 7 secs.	493 MB
JDK 1.6.0	53 mins. 41 secs.	4.13 GB
Apache Project	12 hrs. 50 mins. 21 secs.	28.2 GB

Also, we investigated timing of leveraging the database that contains only JDK 1.6.0. Firstly, we randomly selected several key sequences from JDK source code. Next, we obtained the top 10 value sequences from the database by using the key sequences as queries. We measured the response time of the system, and the average was below 500 milliseconds. The timing does not include timing to display the candidates on the Eclipse plugin. In the implementation, all key sequences are sorted in the order of appearance count. Consequently, timing to inquire into the database had an insignificant effect on database size.

B. Evaluation for Usefulness

When the length of key sequences is small, many value sequences are returned. However, many of the presented code should not be related to code that programmers want. Consequently, we evaluated adequate tokens that programmers should write before triggering a query.

Herein, we assume that a situation to reuse Java source code developed within the same organization. A programmer in the organization remembers the beginning of source code written in the past. The programmer inputs some tokens in his/he memory in order to obtain the past source code.

In this evaluation, firstly, we found out similar methods in different versions from Ant. Then, we investigated whether the proposed technique suggested newer methods from the oldest one. We used 23 versions of Ant, which are from 1.1 to 1.8.2. We detected duplicated code between every consecutive two versions. Then, the detected duplicated code was mapped into Java methods. If the two methods in the consecutive two versions were enough duplicated, they were regarded as reuse instances.

1) Extracting Reuse Instances: If method m_a in version v and method m_b in version $v+1$ are duplicated more than the threshold, we assumed that method m_b had been created by reusing method m_a. In this situation, methods m_a and m_b have a reuse relationship, and we call a pair of methods m_a and m_b a reuse instance.

A duplicated ratio of reuse instance r, which is a pair of methods m_a and m_b, was calculated with the following formula.

$$DuplicatedRatio(r) = \frac{dloc(m_a) + dloc(m_b)}{loc(m_a) + loc(m_b)}$$

where $loc(m)$ is the lines of code of method m, and $dloc(m)$ is the duplicated lines of code of method m.

In this evaluation, we used CCFinderX[1] for detecting reuse relationships. We specified 30 tokens as the minimum

TABLE III
The number of Reuse Instances

Group	Duplicated ratio	# of reuse instances
high	90% or more	32
middle	between 80% and 90%	90
low	between 70% and 80%	211

size duplication to be detected. Also, if $loc(m_a)$ or $loc(m_b)$ is less than 100 tokens, they were not regarded as a reuse instance.

Table III shows duplicated ratio and the number of detected reuse instances. Reuse instances are classified based on their values. Classified groups are called *high*, *middle*, and *low*. There are 32, 90, and 211 reuse instances in *high*, *middle*, and *low* groups, respectively.

2) Detection Result: We wrote the beginning part of method m_i in version v on Eclipse Java Editor, then we checked whether the plugin suggests method m_j in version $v+1$. The input queries were 30, 40, 50, 60, 70, 80, 90, and 100 tokens of the beginning part.

Herein, we assume the followings between version v and $v+1$.

- M_v and M_{v+1} are sets of methods defined in source files of versions v and $v+1$, respectively.
- $r = (m_i, m_j)$ is an instance of reuse relationship, where $m_i \in M_v$ and $m_j \in M_{v+1}$.
- R is a whole set of reuse instances between versions v and $v+1$.
- $RM(m_i)$ is a set of methods having reuse relationships with method m_i, which is defined as follows.

$$RM(m_i) \quad = \quad \{m_j \in M_{v+1} | (m_i, m_j) \in R\}$$

- $Q(m_i)$ is a set of queries generated from method m_i. Herein, $Q(m_i)$ consists of 8 queries, which are 30, 40, $\cdots$, 100 tokens of the beggining part of m_i.
- $SM_{all}(q)$ is a set of all methods suggested triggered by query q.
- $SM_{correct}(q)$ is a correct set of methods suggested triggered by query q. Herein "correct set" is a set satisfying both the following conditions:
  - all the reuse instances related to q are included in it;
  - no method not related to q is included in it.

  It is defined as follows.

$$\begin{aligned} SM_{correct}(q) \quad = \quad & \{m_j \in SM_{all}(q) | \\ & q \in Q(m_i) \wedge m_j \in RM(m_i)\} \end{aligned}$$

By using the above assumptions, precision for group R is defined as the following formula.

$$Precision(R) \quad = \quad \frac{1}{|R|} \sum_{(m_i, m_j) \in R} \left(\frac{1}{|Q(m_i)|} \sum_{q \in Q(m_i)} \frac{|SM_{correct}(q)|}{|SM_{all}(q)|} \right)$$

Also, recall for group R is defined as the following formula.

$$Recall(R) \quad = \quad \frac{1}{|R|} \sum_{(m_i, m_j) \in R} \left(\frac{1}{|Q(m_i)|} \sum_{q \in Q(m_i)} \frac{|SM_{correct}(q)|}{|RM(m_i)|} \right)$$

3) Discussion: Figure 6 shows the result. Both of precision and recall for *high* group are the best in the three groups, and ones for *low* group are the worst. Those imply that the proposed technique works well if the completely-written code of a given half-written method is enough duplicated with a method having reuse relationship with it. For increasing

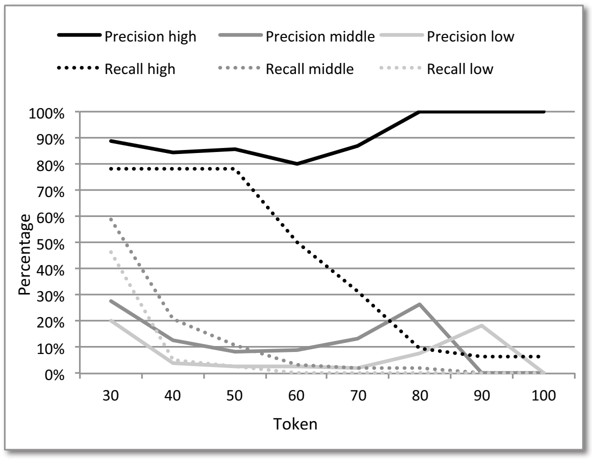


Fig. 6. Precision and Recall

precision and recall on low duplicate reuse, it should be better to use more abstract models on source code such as abstract syntax trees or program dependence graphs. However, those models require much more cost to find reuse relationships than the proposed technique, so that they are not suited for interactive suggestions to users.

The longer queries were, the better precision was. However, recall was opposite. We recommend 30-token queries because of their balance between recall and precision. A reasonable usage should be that: after obtaining multiple suggestions with a 30-tokens query, select one from the suggestions.

For increasing precision, we should analyze source code more deeply to find some heuristics. For example, comparing a list of invoked methods in a query with one of its suggested methods or measuring similarities of code metrics between a query and its suggested methods may be effective.

In this evaluation, the proposed technique was evaluated on method level. However, it can be applied to other levels such as simple block level because it does not depend on any information related to method level.

CCFinderX was used for detecting duplicate code, and 70% or more duplicated methods were used as reuse instances. If we use different detection tools or actual history of copy-and-paste actions, we maybe obtain different results.

IV. Related Work

Grapacc completes the source code under editing based on usage patterns [17]. It extracts context-sensitive features such as data types, the control and branching structures, and the distances of those features with respect to the current editing position. On the other hand, our method helps to complete the source code at any position where a developer would like to complete because all tokens are stored into the corpus.

Hill et al. proposed an approach to complete Java methods as well as our proposed approach [9]. In their approach, firstly each Java method is converted a feature vector comprised of several factors (e.g., the number of lines and the degree of complexity), then the approach suggests Java methods that are neighborhood from a specified Java method and have larger size than the specified one. The suggestion of their approach is a useful if what programmer wants is similar to a specified Java method. However, their approach is unsuitable for our purpose that is Java method completion when programmers wrote only the beginning of a method body.

There are code completion approaches similar to the proposed method. Bruch et al. proposed a code completion method based on existing source code [7]. Robbes et al. proposed a method to use recorded source code histories [18]. Their code completion mechanisms support completion of one sentence. Our approach is to complete the whole method. Their method needs a complete change history of source code. On the contrary, our method requires only source code.

Hipikat seamlessly retrieves source code, problem reports, newsgroup articles that are relevant to developer's current task based on a project memory that consists of project artifacts themselves and also of links between those artifacts indicating relationships [8]. The purpose of Hipikat is seamless retrieval of artifacts but not code completion.

There are several approaches based on API usage information of source code. Prospector retrieves code fragments corresponding to a query that describes desired code in terms of input and output types [14]. Instead of finding code fragments from a database, PARSEWeb uses Google Code Search Engine for gathering relevant code samples [19]. These tools require programmers need to make a query for finding source code. Strathcona retrieves code fragments including API usages based on structural information of source code, in order to support API usage understanding [10]. The tokenization process of our proposed method effectively removes text labels and flattens names for easy retrieval. Both Prospector and Strathcona also ignore text identifiers and concentrate on type tokens in the source code. Michail proposed an approach and tool named CodeWeb to reuse API patterns by using data mining technique[16]. The method only focuses API usage. In contrast, our approach covers all sentences.

The primary strength of the proposed technique compared to existing techniques and systems is realizing seamless code reuse interactively on IDE-based programing.

V. Conclusion

In this paper, we have proposed a technique to support code reuse on demand without suspending coding tasks. The key to search code is half-written code. Users do not have to consider keywords for searching code. Consequently, the time required for code reuse is much shorter than keyword-based code search systems.

Future works will focus on effective transformation rule, partition points between key sequences and value sequences, and key sequences matching rules in the corpus. Furthermore, we plan to develop more suitable ranking methods for providing appropriate reusable code.

Acknowledgment

This study has been supported by the Kayamori Foundation of Informational Science Advancement, Grants-in-Aid for Scientific Research (S) (25220003), Grant-in-Aid for Exploratory Research (24650011) from Japan Society for the Promotion of Science, and Grand-in-Aid for Young Scientists (A) (24680002) from the Ministry of Education, Culture, Sports, Science and Technology.

References

[1] CCFinderX. http://www.ccfinder.net/.
[2] Codase. http://www.codase.com/.
[3] Koders. http://www.koders.com/.
[4] SPARS-J. http://demo.spars.info/.
[5] N. Bettenburg, S. W. Thomas, and A. E. Hassan. Using fuzzy code search to link code fragments in discussions to source code. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 319–328, 2012.
[6] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini. IDE 2.0: Collective Intelligence in Software Development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, pages 53–57, 2010.
[7] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222, 2009.
[8] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.
[9] R. Hill and J. Rideout. Automatic method completion. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 228–235, 2004.
[10] R. Holmes, R. Walker, and G. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. 32(12):952–970, 2006.
[11] M. Ichii, Y. Hayase, R. Yokomori, T. Yamamoto, and K. Inoue. Software component recommendation using collabortive filtering. In *Proceedings of the 31st ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 17–20, 2009.
[12] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 387–391, 2012.
[13] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl.
[14] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.
[15] F. McCarey, M. Cinnéide, and N. Kushmerick. A case study on recommending reusable software components using collaborative filtering. In *Proceedings of the MSR workshop*, pages 117–121, 2004.
[16] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 167–176, 2000.
[17] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 69–79, 2012.
[18] R. Robbes and M. Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.
[19] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, 2007.