Detecting and Visualizing Change Smells Based on Revision History and Code Hunk’s Lifecycle

Woosung JUNG†, Student Member, Eunjoo LEE††, and Chisu WU†, Nonmembers

SUMMARY Change history in project revisions provides helpful information on handling bugs. Existing studies on predicting bugs mainly focus on resulting bug patterns, not these change patterns. When a code hunk is copied onto several files, the set of original and copied hunks often need to be consistently maintained. We assume that it is a normal state when all of hunks survive or die in a specific revision. When partial change occurs on some duplicated hunks, they are regarded as suspicious hunks. Based on these assumptions, suspicious cases can be predicted and the project’s developers can be alerted. In this paper, we propose a practical approach to detect various change smells based on revision history and code hunk tracking. The change smells are suspicious change patterns that can result in potential bugs, such as partial death of hunks, missed refactoring or fix, backward or late change. To detect these change smells, three kinds of hunks – add, delete, and modify – are tracked and analyzed by an automated tool. Several visualized graphs for each type have been suggested to improve the applicability of the proposed technique. We also conducted experiments on large-scale open projects. The case study results show the applicability of the proposed approach.

1. Introduction

Currently, due to the large scale and complicated structure of software applications, it is a difficult and laborious task to detect and fix bugs in software. Many existing researches on automatic bug detection [1]–[7] focus on a single file or module rather than the whole set of changes. For example, the patterns of bug-fix pairs are used to detect potential bugs by using the existing knowledge in the bug-fix memories in a specific project [3]. When developers introduce potential bugs, a fix pattern is recommended using the knowledge existing in the memories. Assume that a buggy hunk H has been fixed as H′, then H′ will be recommended when H occurs in a working revision. In a real work situation, it is often the case that a developer goes back to the H from H′, such as the ‘undo’ action. The change pattern, “H→H→H′”, implies both H and H′ may be problematic code hunks. When the bug-fix patterns in [3] are applied, H′ is recommended when H is re-introduced at the codes; however, this is not correct. That is, it is preferable to consider not only change results such as the bug-fix memories [3], but also the change patterns when predicting bugs.

Additionally, when a code hunk H is copied several times in many revisions/files, most of the copied hunks are often required to be consistent. The consistency requirement means that the code clones should be managed in similar ways. For example, assume that there were three code clones at revision 10 and one of them is removed at revision 20, the remainders are expected to be deleted in the future revisions. There is very little agreement that code clones are consistently handled [8]. Avarsano et al. insisted that code clones tend to be consistently changed [9]; however, Krinke asserted that only fifty percent of them have been consistently handled [10]. In brief, there is no consensus on the consistency of changes in code clones. For all that, it is helpful to alert developers of the codes that should be consistently managed, though the proportion is only fifty percent. “Copy and paste” is a common activity in software development [11], and the rate of duplicated codes is high in an entire software project [11], [12]. Li et al. even persisted that the identifier modification problem is the main cause of bugs in duplicated bugs [11]. In detail, developers often forget to modify the copied identifiers after copy and paste. Conclusively, there is no doubt that code clones should be handled carefully. For example, a test stub is copied onto several files, and parts of the copied stubs become unavailable by comment, after testing. Missed refactoring is another example. When the ‘extract method’ refactoring is applied to several target codes scattered on different files, there can exist codes that refactoring was not applied to due to developer error. In those cases, the stub codes that were not enclosed via a comment symbol, like “/* ... */”, and the remaining methods that should be extracted are code clones and more processing is required on them.

In this paper, a method is suggested to find the change trends in code hunks by tracking code hunks in the views of add, delete, and modify actions, from birth to death of the hunks. The tracking is executed on each revision. The proposed technique does not focus on determining whether or not a code hunk contains bugs like in [3], but the purpose of this work is to show the change trend and several ‘change smells’ in the changes. The ‘change smell’, comparable with ‘code smell’ in refactoring [13], indicates the change trend that goes against the main change trend. The change smell in this work includes partial death of hunks, missed refactoring or fix, and backward or late change. It cannot be said that that code hunks that have the change smell are wrong; however, the hunks are ‘suspicious’ and they are required to be checked by developers for refactoring or fixing the codes. In the test stub example, the test
stubs are add-hunks. The stubs were added as the revision increased, and parts of them are then removed by commenting. In this situation, ‘removing’ is the main trend in the test stub codes and the copied test stub codes that are not removed are considered as ‘partial survival(death)’.

The major assumptions of this paper are as follows. First, the partial death of a hunk is a suspicious state because it could be a potential miss of refactoring or fix that has been done to other hunks except the one that has survived. Second, there exists a trend in changes of small codes. Thus, the modifications that conflict with the major trend bring about a suspicious state. For example, if only one ‘Vector’ class has been changed to a ‘List’ class while ten ‘List’ classes have been changed to a ‘Vector’, then we can smell a bad change there. The first assumption is applied to the add/delete-hunks and the second to the modify-hunks.

In this paper, the change trends are also visualized to help users to check the change smell by defining three kinds of graphs for add, delete, and modify. Several studies have been conducted to visualize the software evolution using repository data [25]–[29]. The goals of the researches are various and different from that of our approach: Girba et al. focused on developers [25]. The general characteristics of software evolution [26], [27] or change related with structural refactorings [28], [29] are shown by using history data. In particular, the ‘change smell’ metaphor is also used in [28], which indicates the structural deficiency. The ‘change smell’ in this paper is closely related with code clones, not the structure of codes. The elements of the graphs in this paper are composed of code hunks, author, version information, and colored edges that indicate different change types. These graphs can be important materials that are included in a project management tool.

Finally, we applied our approach to large-scale open-source projects, JEdit [14], LatexDraw [15], and OSwing [16]. From the experiment results, several graphs have been generated to show the change smells such as potential bugs or refactoring candidates. The proposed approach does not assume any fixed pattern of bugs and can be applied to any programming language. It also can be said that the approach is project-specific because the analysis is conducted based on the revision history and code maintained in a project. Our method takes a similar advantage of code clone detection that finds inconsistent changes [11], [17]–[19]. However, the inconsistent changes identified by the code clone detection tools are limited by clones or rules. Our code smells identify more general patterns in change activities.

The remainder of this paper is organized as follows. Section 2 describes the definitions of the proposed change patterns and detecting techniques for each case, add, delete, and modify. Section 3 explains the processes of add, delete, and modify based analysis. Section 4 shows the results of the case study. Section 5 describes related works, and Sect. 6 outlines the summary, contributions, and limitations of this study.

### 2. Terms and Definitions

The purpose of this study is to support code maintenance by detecting various change smells such as potential miss of refactoring or bug-fix, wrong direction of modification, and so on. These faulty cases mostly come from developers’ mistakes, and it is difficult to find only by analyzing the static patterns coming from the codes. Thus, not only the hunks that are added, deleted and modified in every changes, but also relations of revisions, authors and files should be considered in order to predict the states. The hunks used for the analysis are a kind of AST-based code fragments that can be obtained by calculating the differences between every adjacent file in each change.

#### 2.1 File

**FILE**: A set of every files in all revisions and paths of a project including the files that once existed and were removed. A file, f in FILE, has the following attributes and can be uniquely identified with the revision and path.

- f.Path : a spatial path of file f
- f.Rev : a revision # of file f
- f.Author : an author of f
- f.id : a unique id of file f (=<f.Path, f.Rev>)

f. is a previous version of f in the same path. Thus, the following statement is always true for a given f in FILE, if ‘f.1’ exists.

\[(f.1.<Path>=f.<Path>) \land (f.1.<Rev><f.<Rev>)\]

**Differentiable** f. and f. are differentiable iff f. = f.1.

**Δf** A set of file differences between f.1 and f, if f.1 exists

\[Δf=f−f.1=∪hid, f.1−, f., hA, hD>\]

where

hid is a hunk id, hA is an added hunk, hD is a deleted hunk.

#### 2.2 Hunk

**HUNK**: a set of code fragments obtained from the differences between every pair of differentiable files.

\[H_A (AΔ)=∪hA where hA is an added hunk of AΔ\]

\[H_D (AΔ)=∪hD where hD is a deleted hunk of AΔ\]

**Hunk inclusion**

\[(h∈f) is true iff h∈HUNK can be found in f∈FILE.\]

\[(h∈f) is true iff h∈HUNK cannot be found in f∈FILE.\]

We can categorize the difference cases as following Table 1.

| hA has a value? | hB has a value? | Meaning          |
|----------------|----------------|------------------|
| Yes            | Yes            | Move: if hA=hB   |
|                |                | Modify: if hA≠hB |
| Yes            | No             | Add              |
| No             | Yes            | Delete           |
| No             | No             | N/A              |
2.3 Lifecycle of Hunks

In this subsection, the lifecycle of a code hunk is shown in Fig. 1 and several terms about a hunk’s lifecycle are defined. The graphs visualizing the analysis results are also explained, which are defined for the three types of change: add-hunks, delete-hunks, and modify-hunks. The pattern of the add-hunks graph is generated after the add-hunk analysis.

### 3.1 Add-hunk Based Analysis

The origin of each add-hunk has its originator, and the origin is copied by copiers and the originator. In add-hunk based analysis, the origins per originator are found, and the copiers are then traced for each origin. After that, the location and added time of the copied hunk are tracked to get the propagation state of the add-hunk. The pseudo code for add-hunk analysis is as follows:

```plaintext
for each h in HUNK do
    for each f in FILE do
        if h ∈ f then
            if h is added after the origin of h then it is called a Copy of the hunk. The Authors who copied the hunk are called Copier.
            origin: If a new hunk(h#1) of a code is added at the first time, then the hunk and its author are called Origin and Originator, respectively. Only one originator exists for a hunk.
            copy: If the same hunk is added after the origin of hunk, then it is called a Copy of the hunk. The Authors who copied the hunk are called Copier.
            death: For given h∈HUNK and f∈FILE, If the following proposition(h∈f−1∧(h∉f)) is true, then it is said that h died in f. The deletion or modification of a hunk could cause the death of the hunk.
            survival: If a hunk is found in a file, then the hunk is said to be survived in the file. If the survived hunk had been dead in its history, then we can say that the hunk is revived in the file.
            complete death: If every hunk, including its origin and copies, is dead at a certain revision, then it is said that the hunk is in the state of complete death.
            complete survival: If every hunk, including its origin and copies, survives at a certain revision, then it is said that the hunk is in the state of complete survival.
            partial death/survival: If a part of the same hunks died and some survived, then this is called partial death or partial survival. This is one of the suspicious states.

3. Add, Delete, and Modify-Hunk Based Analysis

In this section, the analysis of a hunk’s history is described based on the definitions in Sect. 2. Hunks are classified into three types; add-hunks, delete-hunks, and modify-hunks. The graphs visualizing the analysis results are also explained, which are defined for the three types of change analysis. The graph examples in this section are part of the results of applying each analysis to open sources JEdit[14] and LatexDraw[15].

- GRAPH: “GRAPH A→B” means to draw an arrow from A to B
- Status[h, f]: A kind of dictionary which has h and f as keys. For example, “Status[ha, f]=survival” means that the state of add hunk ha in file f is survival.
- Status[h]: A kind of dictionary which has h as key. This is mainly used to store the final status.
- survivalCount: A variable denotes the count of survived hunks ha.
- deathCount: A variable denotes the count of dead hunks ha.

```
add-hunks is located at the center of the graph, and the add-hunks are located on the border of the graph as a form of propagated network. The size of circle is directly proportional to the number of origin hunks. The color of edges coming from their originator indicates the final state of each hunk as explained in Table 2.

Figure 2 represents a graph from an add-hunk based analysis for the JEdit project [14]. 7,600 add-hunks are located in circumferences, which are created by their originator at the center of circles. In detail, each originator is located on the center of the circle and the hunks which are at first created by the originator are placed the border linked to the originator. Each hunk is given a unique identifier, which indicates it is the origin hunk. When the hunk is copied, modified, or deleted by some authors, a new add-hunk propagation network is generated from the hunk like Fig. 3. The propagation network can be shown by enlarging the border of the pattern of add-hunks graph. In the add-hunk propagation network, colored edges are created between the origin hunk and the authors and the hunk status nodes are made according to the results of copying, modifying, or deleting by the authors. The hunk status nodes incorporate several information such as the action (add, modify, delete), the author, revision number, and file location as shown in Table 2. It is possible to obtain the contribution degree of each developer from the pattern of add-hunks graph. The border of pattern of add-hunks graph is composed of the propagated networks for the add-hunk, and each add-hunk propagation network shows the lifecycle of each original add-hunk. Figure 3 is an example of the add-hunk propagation network.

• Add-hunk propagation network
The propagation network of add-hunk shows where and by whom the add-hunk is propagated or deleted. In Fig. 3, the hunk ID node, #1256319, is the origin hunk created by the author node, shlomy. The revision ID of the originator is the minimum number. In the dotted circle, the edge color is purple, indicating it is a suspicious case. It is because the text on the purple edge is “1/2” indicating partial survival. As the text on in-edges of ezust and k_satoda et al., is commonly “0/1”, which means that the #1256319 hunk is reused once and then deleted, the color of the edge is red implying complete death state.

The text on in-edges of ezust and k_satoda et al., is commonly “0/1”, which means that the #1256319 hunk is reused once and then deleted. In summary, Fig. 3 shows that #1256319 hunk is copied by six developers including ezust et al., but all copied hunks died excluding the hunk by shlomy at revision 13209 as shown in the hunk status node in the dotted circle, which dies because k_satoda modified the hunks at revision 15295. The hunk marked with a circle in Fig. 3 can be a missed change case because the states of other hunks are currently dead. In the add-hunk based analysis, it is possible to trace the copies and their origin in the entire files at every revision. This method enables the checking of the final state of the hunk using the graphs that show the summarized state information of each hunk. The hunks under partial death/survival state may be suspicious because it is likely that the developers mistakenly did not modify or delete the codes that had survived. It may be a false alarm case, such as refactoring activity that deleted the duplicated codes, but nevertheless it is helpful to examine the mistakes about several changes on codes. This is because checking the whole codes is not required and developers are automatically notified of the information including the location of suspicious hunks. In addition, when the number of survival hunks is too excessive, refactoring is recommended, such as ‘extract method’, though complete survival.
The add-hunk based analysis helps to find suspicious state and refactoring candidate and to get useful information such as the common patterns in codes, files, and revision data. The case study in Sect. 4 shows that a partial death of hunk is highly related to the consistency problems in the maintenance process.

3.2 Delete-hunk Based Analysis

Delete-hunk based analysis finds suspicious revival or survival states of a hunk that has been once deleted. This is similar to the add-hunk based analysis in that both approaches commonly track the lifecycle of each hunk. However, this analysis starts from a delete-hunk pattern and focuses on the sequence and location of the hunk, that is, the revision number and path of the files. The basic idea is that a hunk, which has been deleted at a revision n, is suspicious when the copies of the hunk are inserted and were not then deleted before revision n. The ‘delete’ incorporates not only ‘removing the codes’ but also ‘changing the origin codes by modification’, like add-hunk based analysis.

In order to track a delete-hunk, the algorithm finds every file f, satisfying ‘h not in f1-’ and ‘h in f’ for the given hunk h. Next, it compares the revision of f with a file where the given hunk has been deleted. If the revisions are equal, then the paths of compared files must be different. It is because a hunk that is deleted in a certain file f’ means that f1- must have the hunk, which is contradictory to the satisfied proposition. Thus, the hunk is inferred to be moved from f to f’. As we will examine further in the case study, many of these Move case result in from a refactoring process. Therefore, with this approach, the patterns of a refactoring target can be automatically tracked and gathered even without the hints from the log of repository. Later, the patterns can be applied to find refactoring-missed hunks that might need to be refactored, but missed by mistake. The pseudo code for delete-hunk analysis is as follows:

```
For each hd ∈ HD(∪F:FILE | f ) do
    GRAPH “hd. fnew.Path → hd” with EDGE “hd. fnew.Path”
    Let F = (F ∈ FILE | (hd 2 f1- \f hd≤ (0))
For each f ∈ f do
    GRAPH “hd. f.Path” with EDGE “f.Rev”
    If f.Rev = hd. fnew.Rev_ Then
        Let Status[hd. fnew.Path → hd] = “Moved”
    Else If f.Rev > hd. fnew.Rev_ Then
        Let AfterSet =
            {f′ ∈ FILE | f′.Path = f.Path \ f′.Rev 2 f.Rev, (f′.Path 2 hd 2 f’)}
        Let sucFile = f′
        s.t. ∈ AfterSet, if f′ ∈ AfterSet, f′.Path ≤ f.Rev
        If EXISTS(sucFile) Then
            GRAPH “sucFile.Path → hd” with EDGE “sucFile.Rev”
            Let Status[hd → f.Path, f.Rev] = “Revival”
```

The algorithm considers a birth as a normal case when its revision is earlier than that of the moment of deletion and the paths are identical. However, if the paths are different and the hunk found is still survival, then it is a suspicious case. It is because the survived hunk could be considered missed by mistake when deleting the pattern of the delete-hunk. And the revived hunk after the revision of the given hunk is also considered as a bad change smell if it is survived at the final revision.

After completing the delete-hunk based analysis, a graph is showing the survival patterns for each delete-hunks.

- delete-hunks analysis graph

Figure 4 shows the all change patterns for all delete-hunks of JEdit. It is composed of the subgraphs like Fig. 5. In delete-hunks analysis graph, a specific file becomes a starting point. That is, when a hunk in the specific file was deleted, the deleted hunk is given a unique identifier and
Table 3  Notation in delete-hunks analysis graph.

| entity          | format/property/meaning                                                                 |
|-----------------|-----------------------------------------------------------------------------------------|
| file path node  | <file path> (e.g., /JEdit/trunk/org/gjt/sp/jedit/menu/EnhancedMenuItem.java in Fig. 5) |
| hunk ID node    | hunk’s ID starting “?”                                                                  |
| edge’s color    | • red: the hunk which was deleted is finally survived. (suspicious)                     |
|                 | • blue: a hunk is dead and reused at the same revision, which indicates ‘move’.         |
|                 | • black: a hunk is dead.                                                                  |
|                 | • purple: a dead hunk is revived, but it is finally deleted.                            |
|                 | • brown: a hunk is ‘created’ before deleted.                                              |
|                 | • orange: a hunk is deleted by ‘fixing’ explicitly. (Only red and blue edges should be noticed, and others, which are regular cases, shows just information of the hunk’s history) |
| text on edge    | name@revision                                                                           |
| (from/to hunk’s ID) | name: the developer’s name                  |

Table 4  Notation in overall change graph.

| entity         | format/property/meaning                                                                 |
|----------------|-----------------------------------------------------------------------------------------|
| rectangle      | modified entity (e.g., Vector in Fig. 6)                                               |
| edge’s thickness | it shows the change frequency, so thick edge                                          |
| name on edge   | the number of modification                                                              |

Fig. 6  Examples of OCG.

The analysis process is as follows. First, it sorts the modify-hunks by revision in ascending order. Second, it inserts an old and new node that provide a directed edge from the old one to the new one. Third, when the edge is constructed, two kinds of time stamps are recorded for the old and new nodes. If a new case of modification is inserted (lead, split) or modified to the final nodes (follow) and there is no conflict in the order of time stamps, then there are no suspicious cases. If a conflict of the modify direction with the order of time stamps (back) or the late joining of a previously modified node to other hunks before the edge is constructed, then the (join, jump) are considered as suspicious states. However, we need to compare the change counts in order to reach a more detailed conclusion and reduce false alarms. This is one of the goals of our future work. The change graphs that have connected the edges between old and new hunks are constructed during this process.

In the modify-hunk based analysis, the overall change graph and advanced change graph are created. Table 4 explains the notations used in the overall change graph.

3.3 Modify-hunk Based Analysis

Modify-hunk is a set of hunk pairs <h_D, h_A> in the case of ‘h_D=h_A’, and both hunks exist at the same time. The hunk pairs mostly come from the changes of one node in AST, which provides strong guarantee that the two hunks come from the matching positions of the AST nodes. In this paper, a tree-sized change is considered as a combination of the deletion and addition of the AST subtrees. Thus, it is included in the add/delete-hunk based analysis. The modify-hunk based analysis finds suspicious cases of a file at a specific path, which is different from the add/delete based analysis. The overall view is created with all modify cases to look into the entire modification trend.

The analysis process is as follows. First, it sorts the modify-hunks by revision in ascending order. Second, it inserts an old and new node that provide a directed edge from the old one to the new one. Third, when the edge is constructed, two kinds of time stamps are recorded for the old and new nodes. If a new case of modification is inserted (lead, split) or modified to the final nodes (follow) and there is no conflict in the order of time stamps, then there are no suspicious cases. If a conflict of the modify direction with the order of time stamps (back) or the late joining of a previously modified node to other hunks before the edge is constructed, then the (join, jump) are considered as suspicious states. However, we need to compare the change counts in order to reach a more detailed conclusion and reduce false alarms. This is one of the goals of our future work. The change graphs that have connected the edges between old and new hunks are constructed during this process.

In the modify-hunk based analysis, the overall change graph and advanced change graph are created. Table 4 explains the notations used in the overall change graph.

- Overall Change Graph(OCG)

Figure 6 represents the overall change graphs, which can show the modified contents, change frequencies, and their directions. The thickness of the arrow is proportional to the frequency. From this analysis, the modification trend can be reported and suspicious changes can be automatically detected.

We can observe the class change trends through the revision history. ‘DefaultListModel’ has been changed to ‘Vector’, which has also been changed to ‘LinkedList’, ‘ArrayList’, and ‘java.util.List’ and finally to ‘List’. Some of these have been further changed to ‘ErrorlistForPath’, ‘Collection’ or ‘Set’. We can also observe that ‘Buffer’ has been changed to ‘JEditBuffer’ 210 times, but changes have been made 28 times in the opposite direction. These few changes may have come from bug-fixes, but
they could have also happened by mistake. So, this can be considered as a suspicious case. More detailed traces such as revisions and paths are required to reach a better decision. Table 5 describes the notations used in the advanced change graph.

- Advanced Change Graph (ACG)

The advanced change graphs show more specific information based on revision and author. Figure 7 shows a part of an advanced change graph of PHPParser.java, which is the most changed file in JEdit. The red edge is a change from B to A when the main trend is the change from A to B. The purple edge is a change from A to B after A to B and B to A happened in order. The main trend in this paper is the first change patterns for convenience; however, it would be better to consider the times of modification for accuracy, which will be explored in our future work. In Fig. 7, getShadowColor had been modified into getShadowCol by arn0b at revision 85 six times presented in black edge, and getShadowCol was then changed to getShadowCol at revision 267 three times shown in red edge. After that, at revision 474, getShadowColor was modified to getShadowCol at revision 474 seven times shown in purple edge.

However, the suspicious cases are not always bugs or mistakes. The analysis simply detects change trends based on change patterns. That means that the black or blue edges could be bugs and the red edges could be a case of bug-fixes, or vice versa.

### Table 5  Notation in advanced change graph.

| entity | format/property/meaning |
|--------|-------------------------|
| rectangle | the same as OCG |
| edge’s thickness | the same as OCG |
| edge’s color | • red: suspicious back (if normal trend is “A→B”, the change from B to A is the ‘red’ case)  
• purple: suspicious jump or join (When A is changed into B, and then, B is changed into A. After that, change from A to B is the ‘purple’ case)  
• black: normal case  
• blue: it goes with the normal trend. (The red and purple case should be noticed) |
| text on edge | name(@revision, number)  
name: the developer’s name  
@revision: the revision number  
number: the number of modification |

![Fig. 7 Example of ACG.](image)

Table 6  Summary of change smells.

| Add | Suspectious: A Partially survived hunk is considered as a missed refactor or fix. |
| Refactoring: Hunks in the state of complete survival can be candidates for refactoring if their number is greater than a given threshold. |

| Delete | Suspectious: Revival or survival states of hunks that once had been dead are considered as wrong fix or missed fix. |
| Refactoring: The hunks that have the same pattern but excluded from a refactoring are considered as missed refactoring. |

| Modify | Suspectious: The modifications that conflict with the major trend are considered as backward changes or late changes. |

### Table 6  Summary of change smells.

| Add | Suspectious: A Partially survived hunk is considered as a missed refactor or fix. |
| Refactoring: Hunks in the state of complete survival can be candidates for refactoring if their number is greater than a given threshold. |

| Delete | Suspectious: Revival or survival states of hunks that once had been dead are considered as wrong fix or missed fix. |
| Refactoring: The hunks that have the same pattern but excluded from a refactoring are considered as missed refactoring. |

| Modify | Suspectious: The modifications that conflict with the major trend are considered as backward changes or late changes. |

### 3.4 Discussion

The three types of hunk-based analyses – add, delete, and modify – do not exclusively work, and they may handle some overlapped cases. However, each analysis is mostly performed on distinct parts. For example, delete-hunk based analysis cannot consider the hunks that have not been deleted since being added. In the add-hunk based analysis, the state of add-hunk is traced from birth to death. In the delete-hunk based analysis, the tracking is conducted on the hunks that have been deleted once or more, contrary to the add-hunk cases. Modify-hunks have a feature that its length is much shorter than that of other kinds of hunks. However, the chain graphs that have connected the edges between old and new hunks can be constructed from modify-hunks, which cannot be generated from other hunks. A searching tool such as a hunk finder is necessary for add/delete-hunk analysis. Table 6 briefly explains the change smells that can be detected by our approach.

### 3.5 Implementation

Figure 8 shows the overall activities of the proposed approach which starts from the ‘Sync’ by downloading remote target repository and ends to the ‘Visualizer’ by exporting the graph results. First, a local source repository is synchronized from the remote repository based on the given address of the remote repository. Further, the log data such as author, date, revision or file path are then automatically downloaded and parsed into a local database. And then, source code files are downloaded from the local repository based on the log data. A unique ID is given to each file. Next, the text-based source codes are transformed into XML-based codes.
using ‘Java2XML’ [20] to enable them to be considered as Abstract Syntax Tree’s, which are then indexed in the local database for better searching performance. The file differences, which are composed of add/delete/modify hunks, are calculated based on 'XMLDiff' [21]. The XML-based hunks need to be cleaned up to well-formed XML because they often have mismatched open/closed tags in the process of calculating the differences. Thus, multiple small sub-trees can be obtained by eliminating unnecessary tags and inserting the required closed tags. The hunks are also indexed into the local database by the ‘Indexer’. With the help of a ‘Hunk Finder’, which requires specific queries, various analyses such as detecting clones, tracking the lifecycles of each hunk can be defined and performed. These works are hardly affected by the style of codes compared to those of the text-based methods, because AST is used to detect similar codes in this work. Finally, the specific hunk-based ‘Analyzer’ exports the results after calculating them. And then, they are visualized as a graph in the format of GraphML [22]. The various customized analysis can be applied based on specific conditions such as ‘by author’, ‘by file’ or ‘by date’ because the query options are highly flexible and easily extended.

4. Case Study

In this section, the case study results are presented, which are applied to open source projects JEdit [14], LatexDraw [15], and OSwing [16]. Table 7 shows the statistics of the three open source projects.

| Statistics of open sources. |
|-----------------------------|
| File | JEdit | LatexDraw | OSwing |
|----------|-------|-----------|-------|
| author#  | 108   | 1         | 1     |
| from date(rev#) | 010119(1) | 080308(1) | 070415(1) |
| total hunks# | 437970 | 31918     | 19811 |
| add H (selected H) | 173720 | 11914     | 10697 |
| delete H (selected H) | 172686 | 9077      | 4677  |
| modify H (selected H) | 91564  | 10927     | 4433  |

The hunk selection criteria are as follows: In add/delete-hunks, only the hunks whose lengths are from 500 to 4000 characters after being transformed into XML have been used for analysis. When the size of selected hunks is too short, the selected codes may be meaningless. For example, it is hard to regard a simple function call, like ‘foo(a, b)’, as a meaningful code hunk, though it is copied many times. In addition, precision becomes low due to high false positive rate when the selected hunk size is too small. Therefore, we determined the lower bound as 500 through several experiments. When the size of hunks is too large, the recall rates become low and performance problems arise due to indexing cost. We set the upper threshold to 4000, which is the maximum value not required to be newly indexed. In the modify-hunks, the number of modification is considered at the overview: JEdit selects the hunks with more than ten times, while LatexDraw and OSwing select the hunks with more than five times.

In the following subsections, each case study for add, delete, and modify-hunk based analyses is presented with several suspicious cases and refactoring candidates. We partitioned the revision into half and full (OSwing and LatexDraw), and one third, half, and full (JEdit) to show that the parts of the suspicious cases in the prior revisions have been reflected on the after revisions. The full revision means the whole revisions of the project at the experiment time, and the half revision means the target revisions is from the first revision to the half of the full revisions. For example, if there are total 100 revisions, the target range of half revision is from revision #1 to #50. The numbers of meaningful suspicious delete-hunks and modify-hunks are relatively small in LatexDraw and OSwing, so a comparison between half and full revision has been conducted on the add-hunk based analysis. In case of the modify-hunk based analysis, the results of JEdit are used to show the comparison among revisions.

4.1 Add-Hunks

Figure 9 shows the rates of hunk’s final states of the top 20 originators. The part of ‘partial death’ indicates the proportion of suspicious hunks that each originator created. Seven originators have no suspicious state; however, ruwi and antroy have more than 60% of suspicious states in their origin hunks. This graph is helpful to evaluate the developers’ productivity or their working patterns, etc.

The circumstances are zoomed in at the pattern of the add-hunks graph shown in Fig. 2, then the hunks, copiers, paths and each hunk’s life states could be examined in detail. In Fig. 3, hunk #1256319, which is detected as a suspicious case, is adjust() method, and they are killed by k.satoda. k.satoda inserted code block that is marked in italic and bold at the lower part of Fig. 10. The remaining hunk, which is marked with dotted circle in Fig. 3, does not include the italic code block, which indicates a chance of
Fig. 10 An example of suspicious cases (Add).

Fig. 11 An example of refactoring candidate (Add).

Fig. 12 Examples of files that have different physical paths.

Figure 11 is an example of a detected refactoring candidate in the add-hunk based analysis. The color of every edge is blue, which means complete survival. In detail, the #1294034 add-hunk, which is first created by daleanson at revision 301, is copied by ezust, kpouer, spestov, and daleanson. Figure 11 shows a total of 16 hunks have completely survived. Though several file names are the same, their physical paths are mostly different. Figure 12 presents parts of the physical paths.

Those copies can be the candidates for refactoring like ‘extract method’ because the degree of duplicate codes is relatively high. Actually, the codes in Fig. 13 are commonly inserted in each 16 code hunk that is the refactoring candidate. The codes are all the same except the bold and under-lined ‘52’ that is a trivial token that has little influence on the whole code pattern. The common patterns can cause several problems after code revision, such as, code inconsistency, partial death, and so on.

Next, we compare the results between full revision and half revision to show how the suspicious add-hunks detected in half revision are handled at full revision on LatexDraw and OSwing. Table 8 and Table 9 show the statistics of the half and full revision of each open source.

Table 10 is the comparison results of each open source. When the suspicious hunks that have partially survived at half revision are in the state of complete death or complete survival, it can be assumed that the ‘suspicious’ hunks are actually problematic or refactoring candidates. In case of LatexDraw, 12 add-hunks are completely killed, which is 67% of the partially survived hunks in half revision. In OSwing, one add-hunk is completely dead and one add-hunk has completely survived, which corresponds to 40%.

Figure 14 is an example of completely dead hunks in LatexDraw. The survival result in half is “5/11” which means only 5 hunks has been survived among total 11 copied hunks, and it is then “0/12” in full revision. That indicates that the add-hunk is copied 12 times, but the original and copies are finally completely dead.

The code in Fig. 15 is an example case that “3/4” at
protected void paintBorders(Graphics2D g, Shape s) {
    // Nothing to do.
}
protected void paintBorders(Graphics2D g, Shape s1, Shape s2, Shape s3) {
    // Nothing to do.
}
protected void paintShadow(Graphics2D g, Shape s) {
    // Nothing to do.
}

Fig. 14  An example of actually problematic code hunks in LatexDraw (I).

public void onMouseReleased()
{
    if(action!=null)
    {
        action.end();
        if(action.isMoved())
            UndoCollector.COLLECTOR.add(action);
        action = null;
        gap.setLocation(0, 0);
    }
}

Fig. 15  An example of actually problematic code hunks in LatexDraw (II).

half is changed into “0/4” at full revision. Figure 15 is the old version of a function “onMouseReleased” in four files like MovePtHandler, CtrlPointHandler, ScalsHandler, ArcAngleHandler. In new version, the function is modified according to each usage of each file; therefore, they are completely killed.

4.2 Delete-Hunks

In the delete-hunk based analysis, the hunks with red edges, which indicate a partial survival state, are regarded as suspicious cases. Figure 16 and Fig. 17 illustrate suspicious case examples in JEdit and LatexDraw, respectively. In the two figures, the color of edges in the dotted circle is red, which indicates they are suspicious cases.

The two hunks, #779113 and #779112 in Fig. 16, had been modified or deleted in PHPParserPlugin.java, which was documented as ‘bug fix’ in the log history. After that, the two hunks were reused at revision 11764 by oratherfurd. Before the two hunks had been inserted in PHPParserPlugin.java at revision 8480, #779113 and #779112 were inserted into start() and stopf(), each other. At revision 8480, they are deleted by commenting. However, the comments were released at revision 11754, which is a situation to require the developers’ caution.

In Fig. 17, for two different files whose name is the same as LMenuBar.java, three hunks, #296, #302, and #675 are deleted at one file(‘deleted file’), but they are survived at the other(‘inserted file’). The survived hunks in the ‘inserted file’ may be later deleted, or the deleted hunks in the ‘deleted file’ may be later survived for consistency.

Figure 5 in Sect. 3.2 is an example of refactoring. Two hunks, #711650 and #958056 are the same hunks, used in getShortcut() in two different files, though they have different IDs. At revision 11177, the two hunks are moved into /JEdit/trunk/.../GUIUtilities.java. In the new revision, the private member ‘action’ in the old revision is changed to a parameter, function name is changed, and the access modifier of the function is changed to public static, as is shown in Fig. 18. If one of them had not been moved, it would have been marked with a red edge in the resulting graph, implying missed refactoring.

Figure 19 shows refactoring activities actually conducted in JEdit. As the refactoring messages are logged in the project history, ‘refactor’ was explicitly attached on each edge. In the graph, the color of every edge is blue, which means ‘move’ case. In detail, several hunks in ‘/jEdit/trunk/.../BufferIORequest.java’ are moved onto various files, and it is possible to trace the moved hunks.

The partially survived delete-hunks can be said to be problematic codes when they are finally deleted; therefore, we have compared the results among one third, half, and full
revisions in JEdit for the delete-hunk analysis graphs. We examined how the suspicious hunks in one third and half revisions have been handled at full revision. In case of two third revision, there is little a difference between two third and full revisions; therefore, it is excluded in this paper. Table 11 is the statistics about the comparison results. In Table 11, the ‘suspicious’ row means the number of suspicious cases in each revision, and the ‘deleted’ row means the number of deleted hunks at full revision, which are detected as suspicious at each revision. Thus, the cases of deleted hunks for one third and half are considered at those results.

One of the deleted hunks in half has ‘fix’ log. That is, the suspicious hunk should be buggy. However, only two hunks among 19 suspicious hunks at half revision have been deleted at full revision. In most cases, the consistency between two hunks is expected, so it may be worthwhile to check the ten cases of suspicious hunks in half revision.

One suspicious case in half revision is that a hunk that had been deleted from the fix log was re-inserted. That is, it is helpful to consider the direction of change in that case. The remaining six cases are false positives including two cases that have explicit fix logs. Future work will focus on reducing those false positives in the addition to control or data dependencies.

4.3 Modify-Hunks

Figure 22 shows an example of ACG that presents a change pattern per a developer, ezust, which is intended to present overall trends of modification for a specific author. There is a total of 528 changes, and 20%(106/528) of them are suspicious. However, most of them are concerned with operators such as ‘-’, ‘+’, ‘&&’, ‘|’ , loops ‘for/while’ and booleans ‘true/false’ that can be ignored. After excluding these trivial changes, only 12.7%(67/528) are still suspicious. Figure 23 is the result of enlarging some parts of Fig. 22, as to show the detailed information of the modify-hunks.

Figure 23 (a) shows that ‘EBPlugin’ had been changed to ‘EditPlugin’ during revision 1375 and modified back to ‘EBPlugin’ at the next revision. The path of the file where this change occurred at a specific revision can be easily found with the help of a hunk finder. One of the interesting observations was that a package, ‘org.git.sp.jedit.EditPlugin’, which was imported when EBPlugin was modified to EditPlugin at revision 1375, was left undeleted even after being changed back to ‘EBPlugin’ at revision 1376. That is, the unnecessary ‘import’ remains after the modification, though it does not cause any problems during execution. With the change graph, all the events concerned with a node change such as this case can be traced.
Fig. 23 A partial ACG’s in ezust’s case.

Figure 23 (b) shows that ‘JEditBuffer’ was changed to ‘Buffer’ twice at revision 240, and changed back to ‘JEditBuffer’ at revision 241. Subsequently, ‘Buffer’ was changed to ‘String’ and ‘JEditBuffer’ at revision 1370. Finally, ‘JEditBuffer’ was changed to ‘Buffer’ at revision 9348. In summary, the result is that ‘Buffer’ is changed to ‘String’, though there are several modifications.

We can see a potential problem of this case in the next codes shown in Fig. 24. ‘JEditBuffer buffer = textArea.getTextBuffer();’ can be found in both XmlActions.java and XmlPlugin.java at revision 225. However, the one in XmlActions.java has been changed to ‘Buffer buffer = view.getTextBuffer();’ and the other one in XmlPlugin.java has been changed to ‘Buffer buffer = textArea.getTextBuffer().getBuffer();’ at revision 240, which can be considered as a suspicious state because the same codes were separately changed to different ones.

‘OperatorIds’ were changed to ‘PHPParserConstants’ 16 times and to ‘token2’ 2 times in Fig. 25. However, ‘token2’ was changed to ‘rbrace’ or ‘semicolonToken’ in previous revisions. Thus, compared to the frequency and time order, the purple arrow in the dotted circle appears suspicious in this case, while the other transitions are fine. This could be a potential bug or it could cause a code inconsistency problem during maintenance.

Next, we compare one third revision (referred as part revision) and full revision to check how many the suspicious cases at part revision recurs in full revision. OSwing and LatexDraw have few suspicious cases in half and full revision, so we show only the results of JEdit in Table 12. The ‘recurrence #’ in Table 12 means how many the suspicious cases (red and purple) in part revision reappear in the full revision. As the number is small, modification activity was hardly conducted according to the suspicious trends. Each number for red and purple in Table 12 is the number of edges for each color excluding the modification frequency. However, the ‘recurrence #’ incorporates the frequency, except trivial modification such as operator change, keyword change, etc. For example, when the number of ‘red/one third’ is ten for file A, the file A has ten red edges. In that case, it is likely that the modification was executed once or more for each edge. When the ‘recurrence #’ of red is two for file A, it is possible that either suspicious case happened twice or every two differences suspicious cases happened once. The selected files commonly belong to the upper top ten files ordered by modification frequency, in part revision and full revision.

Table 12 shows that only few suspicious cases in part revision happened in full revision.

5. Related Works

Our work is inspired by code smell by [13]. Code smell indicates suspicion in code such as bug proneness or refactoring candidates. However, our change smell considers the entire change space in the development history and detects suspicious change patterns or activities. Emden et al. proposed a technique to automatically detect and visualize code smell and applied it to the design software inspection tool [23]. They regarded that ‘code smell’ should not be permitted but be used as hints for future refactoring. It is similar to this
work in that ‘smell’ metaphor is used to indicate ‘refactoring’ and visualization technique is provided; however, the purpose of [23] is close to code smell detecting and visualization of Fowler’s refactoring [13]. The result of visualization is also simple compared to this work.

Automatic bug detection research is an active area and many practical techniques have been proposed [1]–[7]. These techniques leverage metrics, code patterns, or developer social networks to predict potential bug locations or bug density. For example, bugMem records previous bug patterns and detects similar patterns in new code. However, the latter approaches focus on bugs on a single file or module rather than considering the whole changes. Kim et al. proposed change classification, which predicts bugs in changes [3]–[5]. However, their approaches also focus on a single change. That is, they predict bugs by classifying the list of individual bugs, not by identifying change trends. For example, when a developer creates codes to bug patterns based on history data, he(she) can be automatically provided with the fix pattern for the created code. The basic assumption of the approaches [3]–[5] is that the bug-fix activity is performed by one direction, and they are applicable when there exists explicit fix log in the bug-fix history data. However, it is possible that the original codes are modified into new code for bug modification, and the new codes then turn back to the original codes. Actually, those happened in the real source codes described in Sect. 4. Furthermore, fix logs can be implicitly required in the approach for bug prediction, which is too limited. It is because many fix activities are executed without fix log in real project history data. In this paper, project history data are used as existing bug prediction studies; however, the bug types are not predetermined at this work. The proposed approach in this work is not dependent on bug-fix log because the differences between revisions are used. The focus of this work is not to predict bugs, but to notify suspicious codes for each of the add, delete, and modify cases by several visualized graphs for checking developers’ mistakes like missed change, and to find refactoring candidate due to code cloning.

Although there have been several studies on visualizing the software evolution [25]–[29], the targets of the studies are various and different from that of our approach. Girba et al. proposed an approach to map changes to developers’ identifiers using CVS log data as to utilize the knowledge of developers [25]. They defined a measure for code ownership and visualized the Ownership Map. In Ownership Map, a specific file and the authors who added, deleted, and modified in the file are presented during evolution. The target in [25] is developers, such as the number of developers, their roles, the developers’ behavioral patterns like ‘Monologue’, ‘Bugfix’, ‘Edit’, and so on. Those differ from our approach which focuses on code hunks. Fernandez et al. investigated the evolution of open source project based on social network analysis using CVS repository [26]. Committer network and module network are created, which are composed of vertices and weighted links. Developers or modules become vertices, and the relation based on the number of common commits are edges. Six characteristics have been suggested in [26], such as ‘degree of a vertex’, ‘weighted degree of a vertex’, ‘distance centrality of a vertex’, etc. Van Rysselberge and Demeyer proposed 2D visualization technique which represents “change relevant information” with a graph where X-axis means file and Y-axis means time, for the log data in CVS repository [27]. In the graph, visual patterns are displayed, such as, unstable components, co-changing entities, design and architectural evolution, and fluctuations in team productivity. The approaches in [26], [27] focused on studying the overall evolution of software through analyzing relations in modules and committers [26] or visualizing some change information [27], which is different from ours concerning the evolution of cloned code hunks. Ratzinger et al. built a graph where nodes and edges represent classes and logical couplings, respectively [28]. The graph can identify ‘change smell’ which means structural deficiency that can be future target to be reengineered. The smell includes ‘man-in-the-middle’ and ‘data container’. Görg and Weißgerber detected and visualized the structural refactoring and local refactoring. Several refactorings including ‘move class, move method, pull up method, push down method’ belong to structural refactoring. Local refactorings include ‘hide method, rename method, add/remove parameter’. Class-hierarchy and package-layout views are provided and different refactorings are shown in different colors. The techniques in [28], [29] are similar to our approach in that they are related to refactorings and especially, ‘change smell’ metaphor is used in [28]. However, while the main targets of [28], [29] are structural parts in codes, our approach focuses on the analyzing the trends of cloned code hunks.

Like our code smells, code clone detection techniques detect some inconsistent changes similar to our code smell patterns by tracking clone changes or inferring change rules. For example, Clone Tracker traces clone change history and identifies inconsistent clone changes [17]. Similarly, LSDiff can also identify inconsistent changes by inferring common change rules from history [18]. However, the inconsistent changes identified by their tools are limited by clones or rules. Our code smells identify more general patterns in change activities. Li et al. proposed CP-Miner that finds bugs about copy-paste [11], [19]. In CP-Miner, frequent subsequence mining algorithm, called CloSpan [24], is used for clone detection, and it focused on the changes of identifiers. For instance, assume that H1 and H2 are the pair of clone hunks and H1 and H2 commonly have identifier tmp at the same locations in both hunks except one location. In detail, H1 has four tmp and H2 has three tmp, and the place where the remaining tmp is located in H1 has temp in H2. And then, it is regarded as inconsistent case, which implies bugs. The programmer’s intention is also considered to filter false positives. That is, the number of different identifier pairs in a clone hunk pair, such as <tmp in H1, temp in H2> is considered. When the number is high, it is regarded as the programmer’s intention. When the number is relatively small, he(she) is thought to forget to change. That resembles
the modify-hunk based analysis in this paper because both CP-Miner and the modify-hunk analysis are mainly concerned with the identifiers’ modification. The differences between the two approaches that modify-hunk based analysis finds suspicious hunks using the modification trends by analyzing the revision history, and CP-Miner finds the suspicious cases with modification frequencies. In this paper, the modification frequencies are also shown in the results, and not only modification trend but also the frequencies will be considered in the future work.

6. Conclusion

In this paper, the technique to detect ‘change smell’, which implies suspicious changes or refactoring candidates, is proposed based on the revision history and code hunk tracking. The types of changes are classified into add, delete, and modify. For each change type, the code hunk tracking has been conducted from birth to death. Based on the analysis of hunks’ final states, various change smells such as partial death of hunks, missed refactoring or fix, backward or late change could be automatically detected. Several visualized graphs for each type have been suggested to improve the applicability of this technique. Those graphs are helpful for project maintenance by showing the lifecycle and propagation of hunks.

The proposed technique has been applied to three open source projects, JEdit [14], LatexDraw [15], and OSwing [16]. In the experimental results, concrete suspicious cases and refactoring candidates are shown for the three change types. A comparison between part and full revisions is also conducted to check how the suspicious cases in part revision are reflected in full revision. In conclusion, we also showed that the alarmed hunks that had been detected as suspicious can have potential problems in real projects. However, the results can have false alarms and need to be eliminated for better accuracy. Thus, we are going to apply more detailed clues from the code patterns, such as modification frequencies that CP-Miner [11], [19] does, data dependency and control dependency. The control and data dependencies can be used in the delete-hunk based analysis, and the modification frequencies can be applied to improve the accuracy of the results in modify-hunk based analysis. We are also going to extend these results in order to enable developers to easily find suspicious codes by showing change trends and listing unusual or unreasonable code changes.

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, 2010-0028148).

References

[1] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, “Predicting fault incidence using software change history,” IEEE Trans. Softw. Eng., vol.26, no.7, pp.653–661, July 2000.
[2] T.M. Khoshgoftaar and E.B. Allen, “Ordering fault-prone software modules,” Software Quality Journal, vol.11, no.1, pp.19–37, May 2003.
[3] S. Kim, K. Pan, and E.J. Whitehead, “Memories of bug fixes,” Proc. 14th ACM SIGSOFT Int’l Symposium on Foundations of Softw. Eng., pp.35–45, Portland, Oregon, USA, Nov. 2006.
[4] S. Kim, T. Zimmermann, E.J. Whitehead, Jr., and A. Zeller, “Predicting faults from cached history,” Proc. 29th Int’l Conf. on Softw. Eng., pp.489–498, Minneapolis, Minnesota, USA, May 2007.
[5] S. Kim and E.J. Whitehead Jr., “Classifying software changes: clean or buggy?,” IEEE Trans. Softw. Eng., vol.34, no.2, pp.181–196, March 2008.
[6] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” Proc. 27th Int’l Conf. on Software Eng., pp.284–292, Saint Louis, Missouri, USA, May 2005.
[7] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, “Predicting the location and number of faults in large software systems,” IEEE Trans. Softw. Eng., vol.31, no.4, pp.340–355, April 2005.
[8] R. Koschke, “Frontiers of software clone management,” Proc. 24th Int’l Conf. on Software Maintenance, pp.119–138, Beijing, China, Sept.-Oct. 2008.
[9] L. Aversano, L. Cerulo, and M.D. Penta, “How clones are maintained: an empirical study,” Proc. 11th European Conf. on Software Maintenance and Reengineering, pp.81–90, Amsterdam, the Netherlands, March 2007.
[10] J. Krinke, “A study of consistent and inconsistent changes to code clones,” Proc. 14th Working Conf. on Reverse Engineering, pp.170–178, Vancouver, Canada, Oct. 2007.
[11] Z. Li, S. Lu, S. Myagmar, and Y. Zou, “CP-Miner: A tool for finding copy-paste and related bugs in operating system code,” Proc. 6th Conf. on Symposium on Operating Systems Design & Impm., pp.289–302, San Francisco, USA, Dec. 2004.
[12] C. Kapser and M.W. Godfrey, "Toward a taxonomy of clones in source code: A case study," Proc. Evolution of Large-scale Industrial Software Applications, pp.67–78, Amsterdam, the Netherlands, Sept. 2003.
[13] M. Fowler, ed., Refactoring. Improving the Design of Existing Code. Addison-Wesley, USA, 1999.
[14] JEdit: http://www.jedit.org/ accessed Dec. 10, 2010.
[15] LatexDraw: http://latexdraw.sourceforge.net/ accessed Dec. 2010.
[16] OSwing: http://oswing.sourceforge.net/, accessed Dec. 10, 2010.
[17] E. Duala-Ekoko and M.P. Robillard, "CloneTracker: tool support for code clone management.," Proc. 30th Int’l. Conf. on Software Maintenance and Reengineering, pp.81–90, Amsterdam, the Netherlands, March 2007.
[18] J. Krinke, “A study of consistent and inconsistent changes to code clones,” Proc. 14th Working Conf. on Reverse Engineering, pp.170–178, Vancouver, Canada, Oct. 2007.
[19] Z. Li, S. Lu, S. Myagmar, and Y. Zou, “CP-Miner: A tool for finding copy-paste and related bugs in large scale code,” IEEE Trans. Softw. Eng., vol.32, no.3, pp.176–192, March 2006.
[20] Java to XML: https://java2xml.dev.java.net/ accessed Dec. 10, 2010.
[21] XMLDiff: http://msdn.microsoft.com/en-us/library/aa302294.aspx, accessed Dec. 10, 2010.
[22] GraphML: http://graphml.graphdrawing.org/ accessed Dec. 10, 2010.
[23] E. van Emden and L. Moonen, “Java quality assurance by detecting code smells,” Proc. 9th Working Conf. on Reverse Eng, pp.97–106, Virginia, USA, Oct. 2002.
[24] X. Yan, J. Han, and R. Afshar, “CloSpan: mining closed sequential patterns in large database,” Proc. SIAM Int’l Conf. on Data Mining, pp.166–177, San Francisco, CA, USA, May 2003.
[25] T. Girba, A. Kuhn, and S. Ducasse, “How developers drive software evolution,” Proc. Int’l Workshop on Principles of Software Evolution, pp.113–122, Lisbon, Portugal, Sept. 2005.
[26] F.L. Lopez, G. Robles, and B.J.M. Gonzalez, “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.

[27] F. Van Rysselbergh, “Studying software evolution information by visualizing the change history,” Proc. 20th Int’l Conf. on Software Maintenance, pp.328–337, Chicago Illinois, USA, Sept. 2004.

[28] J. Ratzinger, M. Fischer, and H. Gall H, “Improving evolvability through refactoring,” Proc. 2nd Int’l Workshop on Mining Software Repositories, pp.69–73, St. Louis, Missouri, USA, May 2005.

[29] C. Görg and P. Weißgerber, “Detecting and visualizing refactorings from software archives,” Proc. 13th Int’l Workshop on Program Comprehension, pp.205–214, St. Louis, Missouri, USA, May 2005.

Woosung Jung received the B.S. degree in Computer Science and Engineering from Seoul National University, Korea, in 2003, and was a researcher in SK UB Care from 1998 to 2002. He is currently under a combined master’s-doctoral program studying Software Engineering in Seoul National University, Korea. He is a member of the Software Engineering Lab. His research interests include the area of software evolution, software architecture and adaptive software system.

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 of School of Computer Science and Engineering at Kyungpook National University. Her current interests include software reengineering, software metrics, web engineering, and software evolution.

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.