Monitor-Based Instant Software Refactoring

Hui Liu, Xue Guo, and Weizhong Shao

Abstract—Software refactoring is an effective method for improvement of software quality while software external behavior remains unchanged. To facilitate software refactoring, a number of tools have been proposed for code smell detection and/or for automatic or semi-automatic refactoring. However, these tools are passive and human driven, thus making software refactoring dependent on developers' spontaneity. As a result, software engineers with little experience in software refactoring might miss a number of potential refactorings or may conduct refactorings later than expected. Few refactorings might result in poor software quality, and delayed refactorings may incur higher refactoring cost. To this end, we propose a monitor-based instant refactoring framework to drive inexperienced software engineers to conduct more refactorings promptly. Changes in the source code are instantly analyzed by a monitor running in the background. If these changes have the potential to introduce code smells, i.e., signs of potential problems in the code that might require refactorings, the monitor invokes corresponding smell detection tools and warns developers to resolve detected smells promptly. Feedback from developers, i.e., whether detected smells have been acknowledged and resolved, is consequently used to optimize smell detection algorithms. The proposed framework has been implemented, evaluated, and compared with the traditional human-driven refactoring tools. Evaluation results suggest that the proposed framework could drive inexperienced engineers to resolve more code smells (by an increase of 140 percent) promptly. The average lifespan of resolved smells was reduced by 92 percent. Results also suggest that the proposed framework could help developers to avoid similar code smells through timely warnings at the early stages of software development, thus reducing the total number of code smells by 51 percent.

Index Terms—Software refactoring, code smell detection, monitor, instant refactoring

1 Introduction

Software refactoring [1], [2] is used to restructure the internal structure of object-oriented software to improve the quality of such software, especially in terms of maintainability, extensibility, and reusability while software external behavior remains unchanged. The word *refactoring* was first proposed by Opdyke [2] and became popular soon after the book by Fowler et al. [3] was published in 1999. However, the core idea of software refactoring could be traced back to *restructuring* [4], [5], which has a long history in the literature. The value of software refactoring has been quantitatively assessed by Kim et al. [6] within Microsoft, and results suggest that the benefit of refactoring is visible.

Tool support is critical for software refactoring [1], [7]. To this end, researchers have proposed tools to facilitate software refactoring. Most mainstream Integrated Development Environments (IDE), such as Eclipse [8], [9], Microsoft Visual Studio [10], and IntelliJ IDEA [11], provide tool support to conduct refactorings on selected source code [12]. Professional refactoring tools have also been developed. However, these refactoring tools cannot be applied until refactoring opportunities are identified by developers or detection tools.

To facilitate the identification of refactoring opportunities, experts have summarized a number of typical situations that may require software refactoring [3], [13], which Fowler calls *Bad Smells* [3, Chapter 3]. Researchers have proposed numerous detection algorithms to identify different kinds of code smells automatically or semi-automatically [14], [15], [16], [17], [18], [19], [20], [21], [22], [23].

However, existing smell detection tools and refactoring tools remain passive and human driven. These tools could not take effect until developers realized that they should refactor, and thus ran these tools. As suggested by Murphy-Hill et al. [24], developers may fail to invoke refactoring tools and smell detection tools as frequently as they should. One of the possible reasons is that they are unaware of the existence of these tools. Another possible reason is that they do not know when these tools should be invoked or that they merely forget to invoke these tools. As a result, human-driven smell detection and refactoring tools fail to drive developers to detect and resolve code smells. Thus, numerous critical refactoring opportunities might not be promptly identified and resolved by inexperienced software engineers.

Some milestones might remind developers to refactor when both smell detection and refactoring tools fail to do so. For example, companies such as Microsoft might restructure an application right after it is released [25]. As an alternative, other companies might conduct refactorings at the beginning of a new iteration. Aside from beginning-of and end-of development iterations, the accomplishment of functional development might also remind engineers to refactor. If functional development is accomplished ahead of schedule, developers tend to improve software quality by

- *H. Liu and X. Guo are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: liuhui08@bit.edu.cn, 630632077@qq.com.*
- *W. Shao is with the Key Laboratory of High Confidence Software Technologies, Peking University, Ministry of Education, Beijing, China, and the Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China. E-mail: wzshao@pku.edu.cn.*

Manuscript received 25 May 2012; revised 9 Oct. 2012; accepted 24 Dec. 2012; published online 9 Jan. 2013.
Recommended for acceptance by M. Robillard.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-05-0139.
Digital Object Identifier no. 10.1109/TSE.2013.4.

refactoring, especially when future maintenance and evolution are predictable.

However, refactorings that are initiated by these milestones are usually delayed, thus resulting in long smell lifespan and high refactoring cost. Before these reminding milestones occur, smells that have been introduced in early stages have already remained for an extended period. Longer smell lifespan results in more expensive smell resolution. This case is similar to that of error correction: Larger delays in error detection and correction will incur higher cost [26]. Instant smell resolution is easier because the involved source code that is newly modified by developers remains fresh in developers' memory. By contrast, when code smells are detected later, developers might have forgotten the code completely. Moreover, instant code smell resolution might facilitate further development because software refactoring improves the extensibility and readability of software applications, thus facilitating further development.

In summary, the traditional human-driven refactoring approach (even with the help of a batch-run smell detector) might result in fewer and delayed refactorings, which might consequently lead to poorer software quality and higher software cost, especially for inexperienced developers. To this end, we propose a monitor-based instant refactoring framework, and initially evaluate this framework against the traditional human-driven refactoring. Instant refactoring is aimed at instant removal of code smells by refactoring once such smells appear. Changes in the source code are analyzed instantly by a monitor running in the background. If these changes have the potential to produce code smells, the monitor invokes corresponding smell detection tools, and warns developers to resolve detected smells promptly. Given that code smells are subjective and that no absolute universal criterion for code smells exists, the monitor collects feedback from developers to adapt smell detection algorithms to certain individuals. The adaption might yield higher precision of code smell detection.

The proposed framework has been evaluated and compared with the traditional human-initiated refactoring approach, where smell detectors are run in batch model. The evaluation was conducted by 20 inexperienced graduate students majoring in computer science. Evaluation results suggest that for inexperienced software engineers, the proposed framework is more effective and efficient in that:

- the number of resolved code smells increased by 140 percent;
- smells' probability of being resolved (ratio of resolved smells to introduced smells) has increased by nearly 500 percent;
- the average smell lifespan has been reduced by 92 percent;
- the number of introduced code smells decreased by 51 percent.

In this paper, we make the following three contributions: First, we highlight the necessity to drive inexperienced software engineers to conduct more refactorings promptly. Second, we propose a monitor-based instant refactoring framework. Finally, we implement and evaluate the proposed framework, and the results suggest that the proposed framework could help inexperienced engineers in removing more code smells promptly.

The remainder of this paper is structured as follows: Section 2 discusses the related work. Section 3 presents a monitor-based instant refactoring framework. Section 4 presents and discusses the evaluation of the proposed framework. Section 5 discusses related issues. Section 6 makes a conclusion.

2 Related Work

2.1 Refactoring Tools

Refactoring tools are widely used to facilitate software refactoring. The market is brisk, and dozens of refactoring tools are available, e.g., Refactoring Browser,[1] JRefactory,[2] IntelliJ IDEA,[3] Eclipse,[4] and Visual Studio.[5]

Existing refactoring tools are essentially similar [27]. Consequently, we explore how refactoring tools work by taking Eclipse as an example. Suppose a developer wants to rename the method *getPrice*. He initially identifies the method, then selects the method, right clicks the mouse, and finally selects the refactoring menu *rename*. The selection would start the refactoring tool, and a refactoring dialog would be proposed. The developer could specify a new name for the method, and the refactoring tool would then change the name and update all references (if selected) automatically.

As described in the preceding paragraphs, existing refactoring tools cannot be invoked before refactoring opportunities (code smells) are identified manually or automatically by independent smell detection tools. In the given example, the developer should manually decide on which method should be renamed before the refactoring of *rename* could be conducted. As a result, software refactoring is essentially human driven. The time and frequency of refactoring depend on software developers.

Researchers are attempting to improve the usability of refactoring tools. Murphy-Hill and Black [27] propose five principles to improve the usability of refactoring tools:

- provide easy ways of selecting desired refactoring;
- switch seamlessly between refactoring and development;
- provide easy ways of exploring source code when refactoring tools are active;
- no explicit configuration requirements;
- never impair accessibility of other tools.

These principles would make refactoring tools more suitable for small-step floss refactoring. But these principles do not change the fact that refactoring tools are human driven.

2.2 Smell Detection

Code smells are signs of potential problems in the code that might require refactorings. Dozens of code smell detection tools have been proposed for the automatic or semi-automatic detection of code smells.

1.http://st-www.cs.illinois.edu/users/brant/Refactory/RefactoringBrowser.html.
2. http://jrefactory.sourceforge.net/.
3. http://www.jetbrains.com/idea/.
4. http://www.eclipse.org/.
5. http://www.microsoft.com/visualstudio/en-us.

Travassos et al. [28] propose a set of reading techniques to help developers in identifying smells. However, the identification is essentially manual. Tourwe and Mens [21] define smell detection algorithms with logic rules in SOUL, a logic programming language. Once compiled, the rules could be readily executed for smell detection. Aside from smell detection, Tourwe and Mens [21] also gather useful information with the rules to propose refactoring suggestions. Similar work is also reported by Moha et al. [20], [19]. The difference is that Moha et al. propose a Domain Specific Language (DSL) to specify bad smells, rather than employing an existing logic programming language. The DSL is based on an in-depth domain analysis of smells, and is thus expected to be powerful and convenient. Specifications in DSL are finally transformed into detection algorithms that can be readily executed. Moha et al. also specify the steps in smell specification and detection, as well as evaluate the approach on open-source systems.

Another category of smell detection algorithms can be described as metric based. Munro [23] proposes a metric-based approach for smell detection in Java source code. Similar work has been conducted by Van Rompaey et al. [22], who propose a metric-based approach for the detection of two test smells: *general fixture* and *eager test*.

Aside from the generic detection algorithms and frameworks that are applicable to all kinds of smells, specially designed algorithms for certain code smells, such as *clone*, have also been proposed to make smell detection more efficient. Code clones are one of the severest code smells, for which a number of algorithms and tools have been proposed [14], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38]. These algorithms have also been evaluated and compared [39], [40]. According to the survey conducted by Zhang et al. [41], approximately 54 percent of studies on code smells focus on code clones, among which 56 percent focus on clone detection.

Incremental detection algorithms for *clone* are also available. Gode and Koschke [42] propose an incremental clone detection algorithm. They generalize suffix trees (a basic structure for clone detection) so that such trees can be incrementally updated, thus enabling incremental construction. Evaluation results presented in [42] suggest that the incremental clone detection algorithm is four times faster than nonincremental algorithms. Hummel et al. [43] propose an index-based clone detection algorithm. The algorithm is not only incremental, but is also distributed and scalable. These incremental clone detection algorithms make the instant detection of clones possible, which consequently makes instant refactoring feasible.

Specially designed detection algorithms for smells other than *clone* are also available. Tsantalis and Chatzigeorgiou [15] propose an algorithm to identify methods that are distributed in wrong locations (*feature envy*), and then suggest how to solve the problems by using the refactoring *move method*. Tsantalis and Chatzigeorgiou [18] also propose an algorithm to detect code smells that could benefit from the employment of polymorphism. Bavota et al. [16], [17] propose an algorithm for the detection of classes that lack cohesion, and then decompose such classes by using *extract class* based on an analysis of relationships among methods. Tsantalis and Chatzigeorgiou [34] propose an algorithm for the detection of long methods, and then attempt to decompose such methods by using *extract method*. More papers on bad smell detection could be found in the surveys by Mens and Touwe [1] and Zhang et al. [41].

Integrated code smell detection tools are also available, e.g., Checkstyle,[6] PMD,[7] and FindBugs.[8] These tools are composed of a number of relatively independent algorithms to identify different code smells. They provide easy access to numerous small code smell detection algorithms.

However, these detection tools, regardless of how generic or smell specific, are passive and human driven. These tools are provided independently, and thus have to be invoked manually by software engineers. Consequently, these tools would not drive developers to refactor. Instead, such tools are driven (invoked) by developers. Software engineers, especially inexperienced ones, might not invoke code smell detection tools on their own initiative [24], which might negatively affect the frequency of software refactoring.

Another challenge in code smell detection is that code smells are subjective. Defining a universal criterion of code smells is challenging, if not impossible [3], [44], [41]. Consequently, smell detection tools usually have a number of parameters (thresholds) that users might manually adapt according to their own view. However, the adaptation is challenging, especially for inexperienced software engineers.

2.3 Extraction of Source Code Change

Fine-grained source code differencing algorithms [45], [46] make the instant detection of changes in source code possible, thus facilitating instant refactoring.

Fluri et al. [45] propose a tree-based differencing algorithm for the extraction of fine-grained source code changes. The algorithm detects changes by finding both matches between nodes of abstract syntax trees as well as a minimum edit script that can make two trees identical. Evaluation results on a benchmark suggest that the proposed framework is effective.

Robbes and Lanza [46] extract fine-grained source changes by hooking IDEs. They also implement a prototype, *SpyWare*, which generates change operations according to events that occur in Squeak IDE.

2.4 Metrics Tools and Code Inspection Tools

Aside from professional metrics tools [47], most mainstream IDEs also provide metrics tools for the quantitative assessment of software quality. Metrics would be updated immediately once the *update* button is clicked, and metrics with values that are out of the predefined normal ranges would be shown in warning colors, e.g., red. As a result, developers would notice the warnings and take necessary measures, e.g., refactoring. Traditional metrics tools are human driven and might thus be incapable of driving instant software refactoring.

Code inspection tools, e.g., ReSharper,[9] are also employed to improve software quality and productivity. These

6. http://www.checkstyle.sourceforge.net.
7. http://www.pmd.sourceforge.net.
8. http://www.findbugs.sourceforge.net.
9. http://www.jetbrains.com/resharper/.

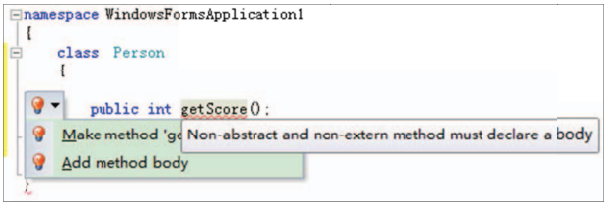

Fig. 1. Code inspection and quick-fix suggestions.

tools are typically monitor based, and are capable of giving suggestions immediately once anything wrong appears. An example of code inspection and quick-fix suggestions is presented in Fig. 1. Method *getScore* is declared, but no implementation is provided. *ReSharper* detects this syntax problem instantly, and then warns developers by underlining the method name. The tool also provides quick fixes (shown as red bulbs in Fig. 1) to fix the detected problem.

However, code inspection tools intend to detect compiler errors and warnings, although some inspection tools also give their own warnings that *"don't prevent your code from compiling but may nevertheless represent serious coding inefficiencies."*[10] Strictly speaking, these warnings are not code smells because code smells are concerned with maintainability and extensibility rather than efficiency. As a conclusion, inspection tools focus on errors (syntactic or semantic) rather than code smells. Thus, in most cases, inspection tools facilitate error fixing rather than software refactoring.

3 Framework

In this section, we propose a monitor-based instant refactoring framework that instantly detects and presents code smells, and drives developers to resolve such code smells.

An overview of the proposed framework is presented in Fig. 2. The framework is made up of a monitor, a set of smell detectors, a set of refactoring tools, a smell view, and a feedback controller. In the following paragraphs, these components will be separately introduced.

3.1 Monitor

The monitor oversees changes made on source code. Once the monitor realizes that such changes might introduce code smells, it invokes corresponding smell detection tools. Consequently, the main task of the monitor is to analyze changes instantly, and then to decide when and which detection tools should be invoked.

A potential challenge that the framework might encounter is that the framework might seriously affect IDEs' performance. Smell detection tools are usually resource consuming. Consequently, IDEs might become unresponsive if smell detection tools run frequently so as to report smells instantly. To minimize this impact, the monitor takes several measures to minimize the frequency of invocation of smell detection tools under the premise of timeliness.

First, the monitor accumulates changes. The monitor decides when to invoke detection tools. In extreme cases, the monitor might invoke detection tools whenever a small character in the source code is changed, which may result in

10. http://www.jetbrains.com/resharper/features/code_analysis.html.

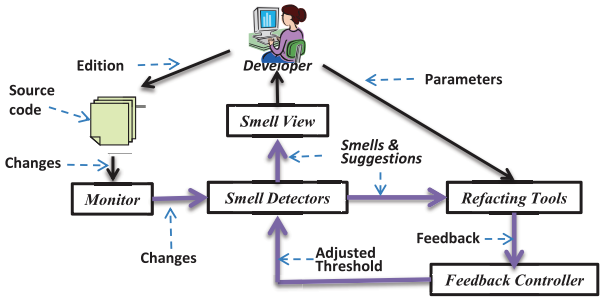

Fig. 2. Overview of the framework.

an extremely high frequency of invocation. To avoid this problem, the monitor employs some kind of buffering wherein changes are accumulated for a while before detection tools are invoked. For most smells, a possible buffering time is the interval between saving (or submitting) actions, i.e., detection tools are invoked only if a file is to be saved or submitted. As a result, invocation frequency is controlled, whereas changes are guaranteed to be fresh in developers' minds. This strategy is adopted in our prototype implementation *InsRefactor*. For some extremely complex detection algorithms, the monitor might employ a longer buffering time.

Second, search scopes are minimized for smell detection tools. Assuming code smells within the unchanged source code have already been identified, detection tools for the instant refactoring framework are incremental, i.e., the tools focus on new smells introduced by the most recent changes. Consequently, the monitor explicitly specifies the most recent changes, and invoked smell detection tools minimize their search scope according to such changes. A narrowed-down search scope significantly improves the efficiency of detection algorithms because the complexity of these algorithms is usually in direct proportion to the search scope.

For some kinds of code smells, e.g., *public field* and *data class*, the search scopes of their detection algorithms are confined to newly modified files. However, this condition is not necessarily true for other smells. For example, to detect *duplicate code* caused by the most recent changes, detection tools have to search the entire source code. However, even in this case, incremental detection tools could be faster than traditional nonincremental detectors. Duplicate code could be classified into three types: those within the unchanged part, those within the changed part, and those between the changed and unchanged parts of the source code. An incremental detector searches for the last two types only by assuming that the first type has already been detected. As a result, search time is significantly reduced because the unchanged part is usually much larger than the changed part.

Third, time consuming smell detection runs in the background; thus such detection would not block the main threads of IDEs. As a result, developers can continue coding while smell detection algorithms are running.

3.2 Smell Detectors and Refactoring Tools

This part contains a set of smell detectors and refactoring tools. The refactoring tools are the same as the existing ones.

However, the detection algorithms are slightly different from existing ones in that:

- These algorithms minimize search scopes to save search time. As discussed in Section 3.1, the monitor explicitly specifies the most recent changes, and smell detection algorithms minimize the search scope according to such changes.
- Some detection algorithms are adapted to take full advantage of the reduced search scope. Usually, the specified search scope would not require essential changes in the smell detection algorithms. However, some existing algorithms might not take full advantage of the minimized search scope without adaptation because the reduction in search scope might become asymmetric. For example, while detecting common methods, incremental detection algorithms compare the changed methods (confined to the changed files) to all methods within an application. On the other hand, existing detection algorithms compare all binary combinations of the methods within the application.

3.3 Feedback Controller

Code smells are essentially subjective. Thus, smell detection algorithms should be optimized according to developers' views. Consequently, detection algorithms for some code smells leave some thresholds for user optimization. For example, to detect the code smell *long parameter list*, detection algorithms should know how long (number of parameters) is "*long*" before any result could be reported. However, the maximum number of parameters (threshold) is difficult to predefine because different people might have different opinions on the maximum number of parameters. As a result, the designers of the detection algorithm leave the initialization of the threshold to users.

However, manual optimization of thresholds is challenging for most software engineers, especially inexperienced ones. Consequently, the proposed framework optimizes thresholds automatically. To this end, the framework initially learns of the user's view, and then identifies the user's opinion on the current value of the threshold by calculating precision, i.e., the number of detected smells divided by the number of smells that have been acknowledged and resolved by the user. Low precision might suggest that the user would like to increase the threshold value, whereas high precision might suggest that the user would like to decrease the threshold value to improve recall. Consequently, we use precision as the feedback for the adjustment of the threshold value by using the feedback controller, as shown in Fig. 3. Recall is not a controlled variable because it is not available. The framework does not have the full list of introduced code smells, and thus could not calculate the recall.

As shown in Fig. 3, the controlled variable is detector precision. The target (reference) is the predefined optimal precision or a predefined range. With actual precision as feedback, the controller updates the threshold (manipulated variable) for the detector. Thus, the actual precision approaches the predefined optimal precision (reference). As a result, the detector achieves optimal precision despite the subjectivity and intuitiveness of code smells.

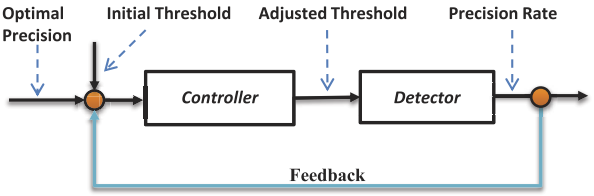

Fig. 3. Feedback controller.

To increase the readability of the framework, feedback control algorithms implemented in the prototype implementation are simple. For example, the feedback control algorithm for *long parameter list* is

$$\text{delta} = (\text{minPrecision}+\text{maxPrecision})/2 - \text{curPrecision};$$
$$\text{threshold} = \text{threshold} * (1 + \text{delta} * 10);$$
$$\text{threshold} = \textbf{max}(\text{minThreshold}, \text{threshold});$$
$$\text{threshold} = \textbf{min}(\text{maxThreshold}, \text{threshold});$$

More advanced feedback control algorithms might speed up the optimization, but might also result in lower readability.

Different control algorithms must be defined for different smell detection algorithms. The thresholds for different detection algorithms have different ranges and different meanings, thus making the design of a universal feedback control algorithm challenging. However, these control algorithms are essentially similar and share the same template, thus facilitating the creation of numerous feedback control algorithms.

Given that the user has a consistent criterion of code smells, the variation in threshold values would become negligible (stable state of control systems) with time. Once the condition does happen, the user might close the feedback controller to improve the performance of the framework.

3.4 Smell View

Detected code smells are presented to developers in a friendly manner. The view is eye catching, such that developers notice it immediately. Moveover, the view is also low key, such that developers can ignore it conveniently (if they will) and continue coding.

We present these smells in a view that is docked around source code editors (the *Smell View* on the right side of Fig. 4). Compared with diagrams, views are low key because their presence or update would not block editors. As a result, developers might ignore such views completely and continue coding as if nothing had happened. To make smell view striking, recently introduced smells are presented in red.

These smells can also be presented by making annotations on the margins of source code editors. This method has been intensively used to remind developers of syntax errors. For example, Eclipse places the mark $\otimes$ on the left margin of Java editors to remind developers that the source code beside the annotation contains a syntax error. By hovering over annotations, developers can obtain detailed explanations and suggestions.

4 Evaluation

4.1 Research Questions

As discussed in Section 1, instant refactoring aims to drive inexperienced software engineers to apply more refactorings

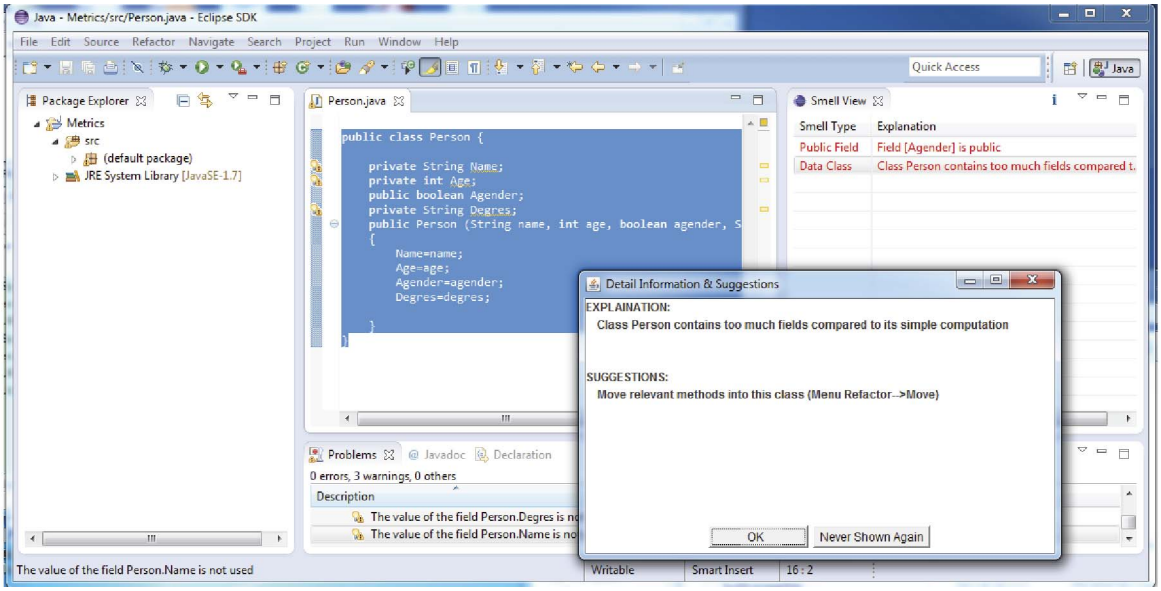

Fig. 4. Prototype implementation *InsRefactor*.

promptly. This goal includes two aspects: 1) more refactorings, and 2) earlier refactorings.

The term *more refactorings* has different but closely related interpretations, e.g., more time spent on refactoring, a larger number of refactorings, and a larger number of resolved smells. If all of the metrics (refactoring time, number of refactorings, and number of resolved code smells) could be collected, the effect of instant refactoring could be validated more thoroughly. However, counting the time and the number of refactorings, especially those that are conducted manually [48], is challenging. Consequently, our evaluation only focuses on the number (absolute number and relative number) of resolved smells, and the results would suggest whether instant refactoring would facilitate a larger number of resolved smells. Special attention is given to the relative number, i.e., the ratio of resolved smells to all introduced smells, because this number suggests how likely it is that the introduced code smells will finally be removed.

The actual meaning of the term *earlier refactorings* is that refactorings are conducted sooner than usual once smells (refactoring opportunities) appear. In other words, the lifespan of smells, i.e., the duration between smells' appearance and disappearance, should be shortened. Consequently, our evaluation would validate whether instant refactoring shortens smells' lifespan.

As a conclusion, the initial evaluation should investigate the following research questions:

- *RQ 1:* Does instant refactoring make IDEs unresponsive?
- *RQ 2:* Does instant refactoring result in a larger number (absolute number and relative number) of resolved smells? If so, by how much?
- *RQ 3:* Does instant refactoring shorten the lifespan of code smells? If so, by how much?
- *RQ 4:* Does the effect of instant refactoring vary with the category (type) of code smells? If so, which kind of code smells would benefit the most from instant refactoring?

4.2 Prototype Implementation

For the initial evaluation of the proposed framework, the instant refactoring framework was implemented as an Eclipse plug-in called *InsRefactor*. A screen capture of the tool is presented in Fig. 4. The / tool, and its source code, are available online.[11]

As a prototype system, *InsRefactor* has implemented detection algorithms for *Data Class*, *Large Class*, *Long Method*, *Switch Statements*, *Public Field*, *Common Methods in Sibling Classes*, *Duplicate Code*, and *Long Parameter List*. *Duplicate Code* detection is based on JCCD[12] [49], an open-source clone detection API. JCCD is written in Java to detect clones in Java source code. Consequently, JCCD can be easily integrated into the prototype implementation, which is based on Eclipse and Java. Modified source code is compared with all other source code within the project. For faster detection, *InsRefactor* caches the pre-processed AST presentation *SourceUnitManager* for each compilation unit.

Identified smells are presented in a "*smell view.*" Smell types, as well as corresponding explanations, are presented in chronological order based on their time of appearance. Developers can locate corresponding source code by double clicking on items (smells) in the view. A double click also brings forward details on smells and corresponding refactoring suggestions. If a developer decides to resolve a selected smell, he can conduct refactorings by employing the imbedded Eclipse refactoring tool or other refactoring tools.

In the prototype implementation, smell detection is run in a separated lower priority thread running in the background. This thread runs in parallel with the Eclipse graphical user interface (GUI) thread. Consequently, developers can continue coding while smell detection is running.

11. http://www.sei.pku.edu.cn/~liuhui04/tools/InsRefactor.htm.
12. http://jccd.sourceforge.net/.

4.3 Setup

4.3.1 Participant and Subject Application

To compare instant refactoring against traditional refactoring (human-driven refactoring with batch-run smell detector), we asked two groups of developers to develop the same software. The experimental group (EG) was equipped with the proposed instant refactoring framework, whereas the control group (CG) was not. All of the involved developers were graduate students majoring in computer science. Through an informal face-to-face survey, we learned that all of the involved developers were familiar with Eclipse, but none of them were experts in software refactoring, which is one of the reasons why they were selected for evaluation. These inexperienced developers are ideal target users of the proposed framework because the framework's primary goal is to drive inexperienced developers to refactor promptly. Participants were randomly assigned to the EG or CG. Each group was comprised of 10 students, but individuals worked independently.

The software to be developed is a Unified Modeling Language (UML) modeling tool, which was selected because the involved developers were familiar with UML and its supporting tools. Consequently, the developers could easily understand the requirements. Submitted applications should support basic class diagrams and use case diagrams.[13] According to our experience, such a system would contain 30 to 50 classes, and 1.5 to 3 KLOC. The average size of the submitted applications was 41 classes and 2.4 KLOC.

4.3.2 Tools

The EG was equipped with *InsRefactor*, whereas the CG was equipped with *InsRefactor′*, a specially designed variation of *InsRefactor*. *InsRefactor′* detects smells instantly as *InsRefactor* does, but *InsRefactor′* would not present code smells to developers. Developers have to fetch the results manually by clicking on a special button, which is similar to traditional smell detection.

Given that traditional batch-run smell detection tools do not optimize thresholds dynamically, *InsRefactor′*, as a simulator of such tools, does not include the feedback controller that was introduced in Section 3.3. To make results comparable, the feedback controller in *InsRefactor* was also disabled in the evaluation.

Smell detection algorithms in *InsRefactor′* are the same as those in *InsRefactor*, which is critical for the subsequent comparison between instant refactoring and traditional refactoring. This characteristic is the reason why *InsRefactor′* was employed instead of other existing smell detection tools.

InsRefactor′, as well as *InsRefactor*, will detect code smells instantly, and will then log the events of smell appearance and disappearance, as well as report statistics, e.g., average lifespan of smells, number of resolved smells, and number of introduced smells. These statistics are critical for the evaluation, which is why *InsRefactor′* detects code smells instantly, although detected code smells are not presented to developers until a *detect smell* button is clicked. If code smells had not been identified promptly, we could not determine their time of appearance nor their lifespan.

13. www.sei.pku.edu.cn/liuhui04/RequirementsOverview.docx.

4.3.3 Process

The evaluation followed the process below:

1. The involved engineers were gathered for a discussion, where they were given the same requirements.
2. Those from the EG were equipped with Eclipse 3.4 and the instant refactoring plug-in *InsRefactor*. On the other hand, those from the CG were equipped with Eclipse 3.4 and a specially designed *InsRefactor* (called *InsRefactor′* for convenience), which was introduced in Section 4.3.2.
3. Each of the involved engineers was given 10 days to develop the system. The participants were told that their submitted applications would be assessed according to both functionality and quality (especially extensibility).

A number of participants may finish the development before the deadline. However, we did not accept submissions in advance because we hoped that they could spend some time on refactoring to improve software quality, which is why the participants were told in advance that some kinds of extension might be required some time later and that their submitted systems would be assessed according to both functionality and quality. To encourage participants to conduct refactoring, we gave them 10 days, although we expected them to finish functional development within 8 to 9 days.

Each of the participants performed the experiment independently. However, the final comparison of instant refactoring and traditional refactoring was based on groups.

Notably, the instant detection of code smells was conducted in both groups. The difference is that detected smells were presented to developers instantly in the EG, whereas in the CG code smells were not presented to developers until a detection command was activated manually (a simulation of traditional on-demand code smell detection).

4.3.4 Measurement

To answer research question *RQ 1*, we collected subjective feedback from participants in group EG (who had employed *InsRefactor* in the evaluation), who would tell whether instant refactoring had made IDEs unresponsive.

To answer research question *RQ 2*, we counted the number of resolved smells. We determined that a smell had been resolved by refactoring if the smell appeared and then disappeared. Aside from the absolute number of resolved smells, we also counted the ratio of resolved smells to all introduced smells, i.e., the relative number of resolved smells. The ratio would suggest how likely code smells are to be finally resolved. We expect instant refactoring to drive developers to resolve most detected smells. Consequently, instant refactoring is expected to increase the ratio of resolved smells to all introduced smells.

To answer research question *RQ 3*, we recorded the appearance time (t_a) and disappearance time (t_d) of smells, and then calculated their lifespan ($lifeSpan = t_d - t_a$). A number of smells might possibly remain unresolved when

TABLE 1
Environment

Items	Description
CPU	Single Dual Core Processor , 2.0 $\sim$ 3.33 GHz
RAM	2 $\sim$ 4 GB
OS	Windows XP, Windows Vista, or Windows 7
Eclipse	3.4

applications were submitted. We ignored such smells while investigating research question *RQ 3* because no corresponding refactorings had been conducted to resolved them.

While investigating research question *RQ 4*, we focused on three aspects of the effect: icrease in smells' probability of being resolved (ISP), reduction in lifespan of resolved code smells (RIL), and reduction in number of introduced code smells (RINI). Consequently, to answer research question *RQ 4*, we computed ISP, RIL, and RINI for each kind of code smell.

4.3.5 Environment

The experimental environments are presented in Table 1. The requirements are not high, and most computers that were assembled in the last three years can satisfy these requirements. We also set the upper limit for hardware performance so that conclusions drawn in this environment, especially those used to answer research question *RQ 1*, are applicable to mainstream computers.

4.4 Results and Analysis

4.4.1 No Serious Effect on IDE Performance

Subjective feedback from developers suggests that instant refactoring had no serious effect on IDE performance (responsiveness). After the experiment, we asked the 10 engineers from the EG (who had employed *InsRefactor*) the following question: *Is the IDE irresponsive?* We provided three choices: "*yes, seriously*," "*yes, a bit*," and "*no, not at all.*" All of the involved engineers selected "*no, not at all.*"

4.4.2 Higher Software Quality

After the experiment, a teacher graded the students' assignments. Assignments were associated with students' IDs instead of names. Consequently, the teacher did not know which group the assignments came from. The evaluation is subjective based on functionality (how well functional requirements were satisfied) and quality (readability and extensibility) of submitted software applications.[14]

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Experimental Group	10	924	92.4	8.0444444
Control Group	10	891	89.1	10.988889

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	54.45	1	54.45	5.7215412	0.027882	4.4138734
Within Groups	171.3	18	9.516667			
Total	225.75	19				

Fig. 5. ANOVA on software quality.

TABLE 2
Results

Items	EG	CG
Resolved Smells	325	135
Increase in Resolved Smells	140%	/
Detected Smells	409	840
Reduction in Detected Smells	51%	/
Resolved Smells/Detected Smells	80%	16%
Average Lifespan of Resolved Smells (hours)	7.7	98
Reduction in Average Lifespan	92%	/

The results are presented in Fig. 5. The results suggest that the EG has achieved higher grades. The average increased from 89.1 to 92.4. The analysis of variance in Fig. 5 ($F > F_{crit}$ and P-value < 0.05, where $\alpha = 0.05$) also suggests that the factor (instant refactoring framework) affected the students'grades.

4.4.3 Larger Number of Resolved Smells

As discussed in Section 4.3.4, we counted the number of resolved smells, the number of detected smells, and the average lifespan of smells. These data are presented in Table 2. The second and third columns of the table present data from the EG and CG, respectively. Detailed metrics for each kind of involved smell are also presented in Figs. 6, 7, and 8. From the evaluation results, several observations can be made.

First, Table 2 shows that the EG resolved more smells than the CG. The EG resolved 325 code smells, whereas the CG resolved only 135 code smells, an increase of up to $140\% = (325 - 135)/135$. From Fig. 6, we also observe that as far as the number of resolved smells is concerned, the EG outperformed the CG on six (out of eight) categories of code smells.

Second, from Table 2 and Fig. 7 we observe that the EG also has a larger relative number of resolved smells, i.e., a larger ratio of resolved smells to introduced smells. The EG resolved 80 percent of introduced code smells, whereas the CG resolved only 16 percent of introduced code smells. In other words, instant refactoring has increased smells' probability of being resolved by $400\% = (80 - 16)/16$.

Third, from Table 2 we observe that the EG has introduced fewer smells than the CG. The EG has introduced 409 code smells, whereas the CG has introduced 840 code smells, a reduction of up to $51\% = (840 - 409)/840$. This finding suggests that instant refactoring might help reduce code smells. One possible reason is that if developers are frequently warned against code smells at the early stage of software development, they would learn to avoid similar code smells in subsequent development.

Finally, from Fig. 6, we observe that the EG has resolved fewer *switch statements* and *common methods in sibling classes* than the CG, whereas the former has resolved more code

14. To make the grading system fair for both groups, the raw grades were subsequently normalized. In this paper, we use the raw grades only.

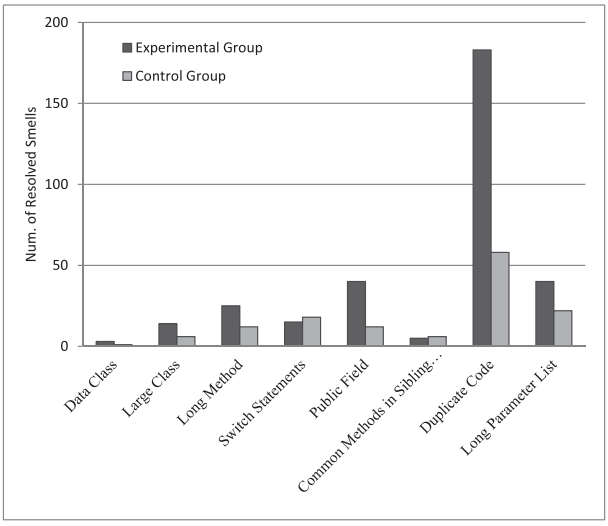

Fig. 6. Number of resolved smells.

smells of other types than the latter. The reasons are as follows: On the one hand, instant refactoring has helped the EG to avoid a number of code smells. As a result, the total number of *switch statements* and *common methods in sibling classes* introduced by the EG is smaller than that introduced by the CG (as shown in Fig. 8). On the other hand, both the EG and CG took *switch statements* and *common methods in sibling classes* as the severest code smells, and both resolved all instances of these two kinds of smells. In other words, the smells' probability of being resolved is 100 percent (as shown in Fig. 7). As a result, the total number of *switch statements* and *common methods in sibling classes* resolved by the EG (*number of introduced smells**100 percent) is smaller than that resolved by the CG.

As shown in Fig. 8, instant refactoring can help reduce the number of introduced code smells. One possible reason is that by being continuously warned by the instant refactoring framework, developers learn to avoid similar code smells. For example, inexperienced engineers might initially define a large number of public fields because such fields are easier to access than private ones. However, once warned numerous times by the instant refactoring framework, the engineers may realize that defining public fields is not a good practice, although they may not

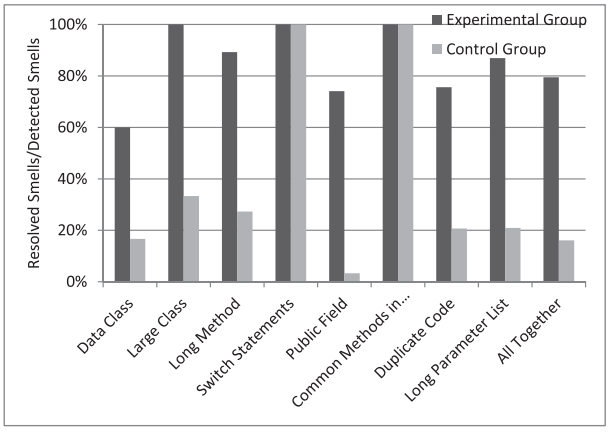

Fig. 7. Smells' probability of being resolved.

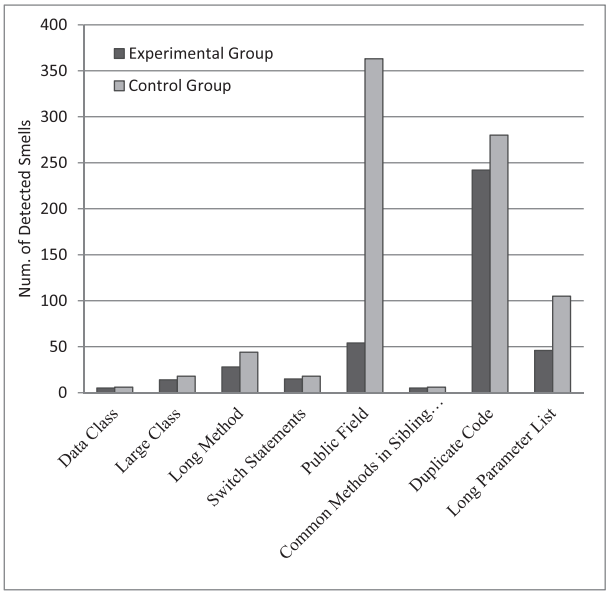

Fig. 8. Number of detected smells.

necessarily understand the reason. As a result, the engineers will avoid public fields in future development whenever possible. Some engineers might also turn to experienced experts for help, i.e., asking for professional explanations of code smells.

According to the discussion in the preceding paragraphs, we might draw the following conclusion: Instant refactoring might drive developers to resolve more code smells (an increase of 140 percent), and may help developers avoid similar code smells by issuing timely warnings (a decrease of 51 percent). As a result, instant refactoring increases code smells' probability of being resolved (an increase of 400 percent). Consequently, the number of code smells remaining in the submitted (released) applications would be reduced by instant refactoring (by $88\% = 1 - (409 - 325)/(840 - 135)$), which might help improve software quality.

4.4.4 Shorter Lifespan of Resolved Smells

The average lifespan of resolved smells is presented in Table 2, and the average lifespan of each kind of code smell is presented and compared in Fig. 9.

From Table 2 and Fig. 9, we observe that compared with the CG, the EG has a shorter average lifespan of resolved smells, i.e., shorter average interval of time between smell's appearance and disappearance. The average lifespan has been reduced from 98 to 7.7 hours, a reduction of up to $92\% = (98 - 7.7)/98$. From Fig. 9, we also observe that the reduction in lifespan was achieved on all kinds of code smells.

Distributions of lifespan are presented and compared in Fig. 10. In the beanplot of Fig. 10, the individual observations (resolved smells) are shown as small lines. If two or more individuals are located in the same place (with equal lifespan), the length would be accumulated. The shape is the density, and the long bold line is the average. As shown in the figure, most code smells resolved by the EG had a lifespan that is shorter than one hour. On the other hand, however, a small number of smells resolved by

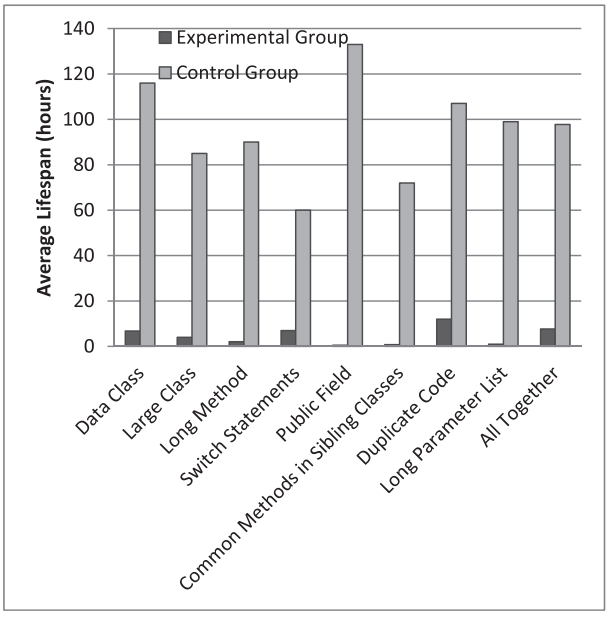

Fig. 9. Average lifespan of resolved smells.

the EG had a longer lifespan (approximately 100 hours). By contrast, most code smells resolved by the CG had a longer lifespan (approximately 100 hours). Only a small number of the smells resolved by the CG had a lifespan shorter than 10 hours.

The EG, the members of which were equipped with the instant refactoring framework, was expected to resolve smells instantly. Consequently, the lifespan of resolved smells should be slightly longer than the time required for smell resolution, which is usually less than one hour. However, the experimental results suggest that the actual lifespan of smells (7.7 hours) is longer than expected. One possible reason is that a number of less severe smells remained unresolved until spare time was available, i.e., functional requirements have been satisfied ahead of schedule, as discussed in Section 4.3.3. When developers identify a less severe (but challenging to remove) code smell, they might ignore such a smell because the schedule is tough. However, if functional requirements have been satisfied ahead of schedule, developers might turn back to resolve the remaining smells, which would result in the smells' long lifespan. The results presented in Fig. 9 confirm that *duplicate code* has the longest lifespan (the EG). If duplicate fragments are short and stable, their negative impact is not severe but their resolution is usually time consuming. Consequently, developers might put off the resolution of such fragments until spare time is available. The distribution in Fig. 10 also confirms this assumption. Another possible reason is that the resolution of some kinds of code smells is time consuming. If code smells are resolved instantly, their lifespan should be approximately equal to the time spent on smell resolution. Consequently, the more complex and time consuming the resolution is, the longer the code smells' lifespan would be. The results presented in Fig. 9, where *switch statement* has the second longest lifespan (the EG), confirm this assumption. Compared with other smells listed in Fig. 9, *switch statement* is more challenging to remove. Usually, the

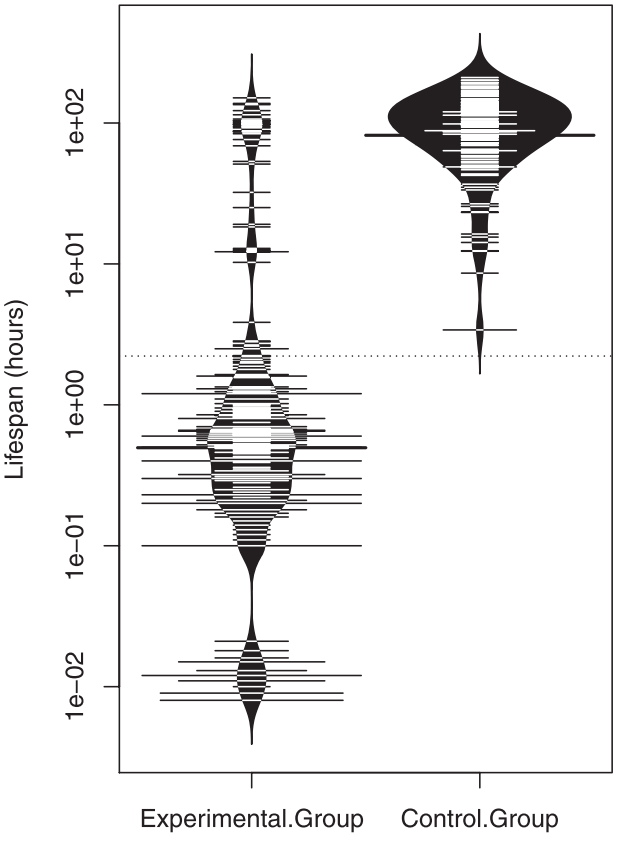

Fig. 10. Lifespan of resolved smells (beanplot).

removal might involve the decomposition of dozens of methods, followed by the redistribution of these methods into a new hierarchy. Consequently, developers may take a few hours to design an appropriate solution and to perform the necessary refactorings.

The results of single factor analysis of variance ($\alpha = 0.05$) on lifespan are presented in Fig. 11. As suggested by the results, the *P-value* is extremely small (1.54773E-94), and F is considerably greater than F_{crit}. All of these findings suggest that the factor (instant refactoring framework) has an effect on the measurement (lifespan of resolved smells). The standardized effect size (Cohen's d) is also calculated as follows:

$$d = \frac{\bar{x_1} - \bar{x_2}}{s} = \frac{98.36 - 7.67}{53.13} = 1.71, \quad (1)$$

where d is larger than 0.8, suggesting that it is a large effect size. The large effect size suggests that the factor (instant

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Experimental Group	325	2493.9	7.673538	640.5767643
Control Group	135	13278	98.35556	2265.664726

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	784335.1	1	784335.1	702.7845407	1.55E-94	3.861842
Within Groups	511145.9	458	1116.039			
Total	1295481	459				

Fig. 11. ANOVA on the lifespan of resolved smells.

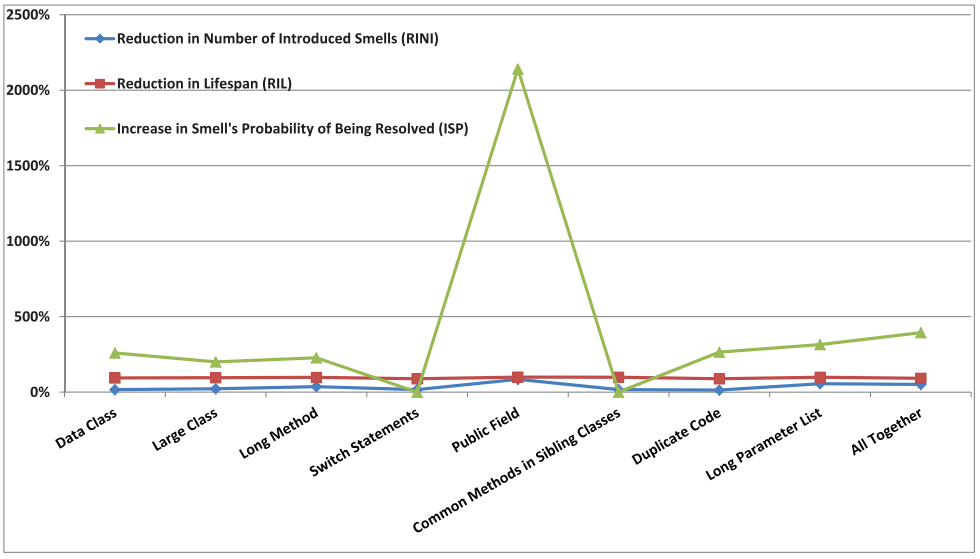

Fig. 12. Instant refactoring's effect on different types of code smells.

refactoring framework) has an effect on the measurement (lifespan of resolved smells).

4.4.5 Different Effects on Different Types of Code Smells

As discussed in Sections 4.4.3 and 4.4.4, the effect of instant refactoring might vary among different types of code smells. To investigate this issue, we present the variation in Fig. 12. The figure focuses on three aspects of the effect: increase in smells' probability of being resolved (ISP, the ratio of resolved smells to all introduced smells), reduction in lifespan of resolved code smells (RIL), and reduction in number of introduced code smells (RINI). From Fig. 12, we might make the following observations.

First, we observe that the variation of RIL is smaller than that of the other two aspects (RINI and ISP). The reduction in smells' lifespan varies from 88 to 99 percent, suggesting that instant refactoring would facilitate earlier smell resolution regardless of the type of code smell.

Second, we observe that instant refactoring has a weaker effect (especially RINI and ISP) on some types of smells, e.g., *switch statements* and *common methods in sibling classes*. One possible reason is that these two types of code smells are more severe than others. As a result, even without any warning developers are aware that they should try to avoid these kinds of code smells, thus weakening the effect on RINI. For the same reason, developers would resolve all *switch statements* and *common methods in sibling classes* without hesitation. Results in Fig. 7 confirm that the CG has resolved all detected *switch statements* and *common methods in sibling classes*, making it impossible for the EG to achieve any improvement.

Finally, we also observe that instant refactoring achieved the greatest effect on *public field*. One possible reason is that *public field* is less severe and easier to remove. The reason why instant refactoring has a greater effect on less severe code smells has been discussed in the preceding paragraph. Given that *public field* is easy to remove, developers are willing to remove them once warned by the instant refactoring framework, which results in higher probability of being resolved. The results suggest that instant refactoring is suitable for smells that are less severe and easier to remove.

The variation of the effect on different code smells except *public field* is presented in Fig. 13. In this figure, we observe that the variation is still obvious, especially the increase in smell's probability of being resolved and the reduction in the number of introduced smells.

4.5 Threats to Validity

The employment of different participants in instant refactoring and traditional refactoring presents a threat to internal validity. The differences of the participants, as well as the differences in the refactoring method (instant refactoring or traditional refactoring), might result in differences in refactoring frequency and smell lifespan. To minimize the threat, we randomly assigned 20 developers to the EG and CG, and each of the developers performed the experiment independently. The final comparison is based on groups (each group had 10 members) instead of individuals, which might help reduce the impact of the participants' individual differences.

Another threat to internal validity is that if participants had known the expectation of the evaluation in advance, participants equipped with the instant refactoring framework might tend to refactor more frequently, whereas those employing traditional refactoring might tend to avoid refactoring. To moderate this threat, we had not told them about the purpose of the evaluation.

A third threat to external validity is that participants in the evaluation were graduate students majoring in computer science. Conclusions drawn from graduates are not necessarily true for experienced software engineers. We selected these graduate students for the following reasons: First, the target users of instant refactoring are inexperienced developers, similar to the participants in our evaluation. In other words, the selected participants are qualified target users of the proposed framework. Second, gathering 20 experienced software engineers for an academic experiment for 10 days is challenging.

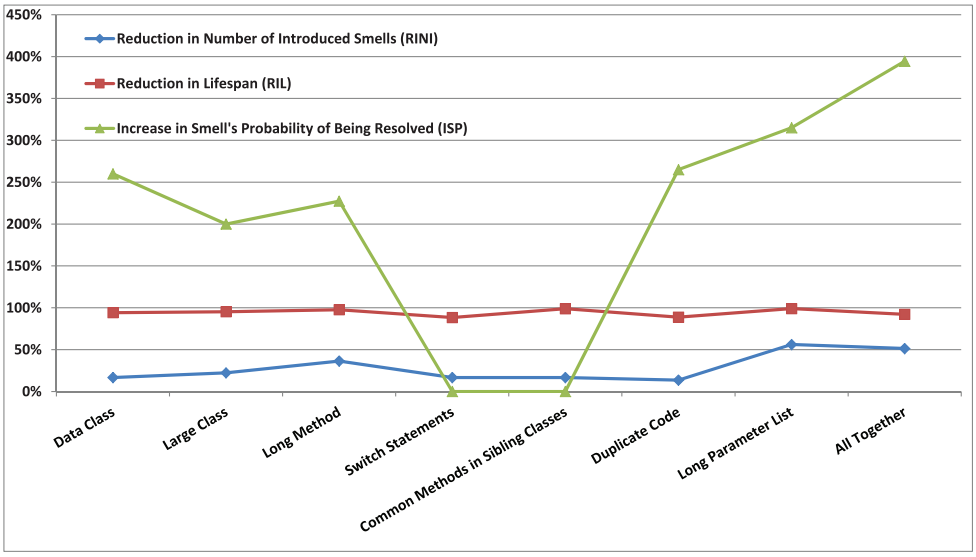

Fig. 13. Instant refactoring's effect on different types of code smells (except *public field*).

A fourth threat to external validity is that the experiment is conducted on a single application. Special characteristics of the application, rather than the difference in refactoring manner, may have yielded the evaluation results. The reason why this application was selected was discussed in Section 4.3.1. To make the conclusions more reasonable, further evaluation on more applications should be done in the future.

A fifth threat to external validity is that only eight types of code smells have been monitored by the prototype implementation that was used in the evaluation. As discussed in Section 4.4.5, the effect of instant refactoring might vary according to types of code smells. Consequently, the conclusions drawn on these eight types of code smells might not hold for other types of code smells. These eight types have been implemented in the prototype for the following reasons: First, these smells are popular. All of these eight smells are among the most popular code smells listed by Fowler et al. [3]. Second, these code smells are easier to detect than others, which made the implementation of the prototype easier.

Another threat to external validity is that in the experiment, developers were given plenty of time to develop and to refactor software applications. However, in real industry developers might have tougher schedules, which might result in fewer refactorings regardless of whether an instant refactoring framework is available. As suggested by the evaluation results, the CG conducted most refactorings after the accomplishment of functional development. Consequently, a tougher schedule would dramatically reduce the number of refactorings conducted by the CG. For the EG, the distribution of refactorings is better balanced. However, even if driven by an instant refactoring framework, developers might hesitate to refactor because of a tougher schedule if they do not know the fact that "*the overhead of refactoring is more than compensated by reduced efforts and intervals in other phases of program development*" [3, Chapter 13 by William Opdyke].

A threat to construct validity is that while measuring smell lifespan, we have not distinguished working time from nonworking time. If nonworking time had been excluded, the lifespan would be reduced. However, distinguishing working time from nonworking time is difficult. The time when developers are coding is sure to be working time. However, the time when developers are not coding is not necessarily nonworking time because these developers might be thinking, planning, or preparing documents.

A second threat to construct validity is that counting resolved code smells based on their disappearance might be inaccurate. The experiment was based on the assumption that if code smells appeared and then disappeared, they have been resolved by refactoring. However, in rare cases, code smells might disappear for other reasons. For example, a *data class* might become a normal class when new operations are added (instead of moving other operations to this class, as suggested by refactoring guidelines). In other words, the smell *data class* disappears because of the addition of new features instead of by software refactoring. However, the smell is still counted as a *resolved* smell in the evaluation because it has disappeared. This threat might be removed by directly counting applied refactorings. However, the identification of refactorings, especially manually applied refactorings, is challenging [48].

A third threat to construct validity is that the logged appearance time (and disappearance time as well) of code smells is an approximation. The reflected time is actually the discovery time of code smells instead of the actual appearance time. Given that the instant refactoring monitor is not real time, logging the actual appearance time is impossible. Consequently, in the experiment we logged smells' discovery time as an approximation of their appearance time. The range of error is the interval between consecutive invocations of smell detection tools.

5 Discussion

5.1 Target Users

The target users of the proposed instant refactoring are developers with little or no experience in software refactoring. With timely warnings, instant monitors drive inexperienced developers to resolve code smells promptly. These

inexperienced developers need the reminders because they lack consciousness of the urgency of smell resolution. How inexperienced developers benefit from instant refactoring has been experimentally evaluated in Section 4.

The value of instant refactoring seems insignificant because its target users (inexperienced engineers) do not make significant contributions to refactoring compared with experts or mid-level professionals. However, instant refactoring helps train inexperienced engineers into experts or mid-level professionals so that they would become a strong force in this area.

For experienced engineers, the proposed monitor is also helpful, although they might benefit less from the framework than inexperienced developers. Focusing on the addition of functional features, even experienced engineers might forget to detect and/or to resolve code smells. In this case, the instant refactoring framework takes effect.

For experienced engineers, the proposed framework could be more effective with slight modification. Assuming that most code smells could be identified by experienced engineers promptly on their own initiative, we might extend the interval between successive invocations of smell detection tools. Giving experienced engineers sufficient time to identify and to resolve code smells would yield fewer disturbing warnings. The reduction in invocation frequency also saves computing resources.

5.2 Extensibility

As a prototype implementation, we have implemented detection algorithms for eight types of code smells. However, numerous code smells have not yet been detected by the prototype implementation. Consequently, the extensibility of the framework is essential.

Smell detectors within the framework are essentially independent of one another. Thus, new detection tools could be easily added. New detection tools should provide the interfaces required by *monitor* and *smell view*. In the prototype implementation, we have defined these interfaces. The prototype also defines a super class *SmellDetector*, which implements the required interfaces. Consequently, the only requirement on new smell detection tools is that they should inherit class *SmellDetector* directly or indirectly.

As discussed in Section 2.2, automatic generation of smell detection algorithms is theoretically possible. However, for complex code smells, e.g., duplicate code, generating detection algorithms is difficult, or the generated algorithms may be less efficient than those delicately designed by experts. Consequently, experts in the real industry must manually design these algorithms, regardless of whether such algorithms run in batch model (traditional refactoring) or in real-time incremental fashion (instant refactoring).

Different controllers should be defined manually for different smell detection algorithms. However, these controllers are essentially similar. With slight modification (changes on thresholds or other constant variables), we can generate a new controller for a new detection algorithm.

5.3 Refactoring Earlier versus Refactoring Later

The monitor-based instant refactoring proposed in this paper encourages developers to refactor earlier, i.e., to resolve code smells instantly once they appear. However, in some cases, the instant resolution of code smells might result in pointless refactorings. For example, all refactorings conducted on a class turn out to be pointless when the class is removed for some reason, e.g., changes in requirements. Delayed refactoring would reduce the probability of pointless refactorings, but may lead to higher refactoring cost, as discussed in Section 1. Any refactoring, regardless of whether it is conducted earlier or later, may theoretically become pointless as long as the requirements and/or source code are exposed to change.

5.4 Warning versus Training

Several methods can be employed to drive software engineers to resolve code smells. The proposed instant refactoring warns developers against code smells during software development. Thus, developers are reminded to clean code smells promptly. An alternative approach is training before software development is initiated. By informing developers of the importance of software refactoring (and some skills in detecting and resolving code smells as well), we might expect developers to perform more refactorings. However, as discussed in the third paragraph of Section 5.1, even if developers are willing to refactor, the timely identification of code smells remains challenging, thus weakening the effect of training. By contrast, the proposed framework automatically identifies code smells and warns developers promptly, thus freeing developers from challenging smell detection.

5.5 Static versus Dynamic Optimization of Thresholds

Defining thresholds for smell detection algorithms by using static analysis is possible. For example, we can discover the mean value (or the upper quartile) of the number of parameters within an application, e.g., a benchmark, and set such value as the threshold for detection of *long parameter*. However, thresholds inferred in this way cannot reflect developers' unique personality, although smell detection is essentially subjective. In other words, different developers may prefer different thresholds for the same smell detection algorithm. Furthermore, developers might change their view over time, making statically inferred thresholds outdated.

Feedback-based threshold optimization for smell detection is personalized and dynamic. Such a process optimizes thresholds by reviewing the developer's ongoing refactoring history. Consequently, the optimization reflects the developer's current opinion.

5.6 Detection of Bad Names

As suggested by Murphy-Hill et al. [24], the *rename* refactoring tool is used more frequently than other tools. This finding might suggest that *bad name* is one of the most basic code smells. However, automatic detection of bad names is difficult, if not impossible, because such detection involves decision criteria that are not easily automated. A name is bad if it does not convey the intent of the source code entity (a class, a field or a method). However, automatically discovering the intent of source code entities

remains challenging. As a result, determining whether a name conveys the intent of a source code entity is difficult.

InsRefactor reports names comprised of only a single character, e.g., x, as bad names.[15] These names are easy to detect, but such names comprise merely a small subset of bad names.

6 Conclusion and Future Work

In this paper, we first highlight the necessity of driving inexperienced software engineers to resolve more code smells promptly. We then propose a monitor-based instant refactoring framework. We also evaluate the proposed framework, and the results suggest that this framework could drive inexperienced engineers to resolve more code smells (an increase of 140 percent) promptly (smell lifespan reduction of 92 percent). Moreover, evaluation results also suggest that the proposed framework could help avoid a number of (approximately 51 percent) code smells.

Future work is needed to extend the framework to deal with more types of code smells. To date, only eight types of code smells could be detected by the prototype. To increase the framework's applicability in the industry, we should design detection tools for other types of code smells, and then integrate such tools into the framework. The high extensibility of the framework, as discussed in Section 5.2, would facilitate such extension. Integrating existing incremental smell detection algorithms into the framework is another way to increase the framework's applicability.

Future work is needed to improve the performance of the framework. As a prototype implementation, *InsRefactor* focuses on functionality rather than performance. However, to increase the framework's applicability in the industry, the framework, especially the resource-consuming smell detection component, should be restructured for higher performance.

Future work is also needed to evaluate the proposed framework further. The proposed framework should be evaluated based on more applications by more software engineers, both inexperienced and experienced. The ultimate goal of the proposed framework is to reduce software cost (especially maintenance cost) and to improve software quality. Although removing more code smells by refactoring is believed to improve software quality [3], [50], and earlier resolution of code smells is thought to save time (as discussed in Section 1), directly investigating the extent to which the proposed framework could improve software quality and reduce software cost remains important.

Acknowledgments

The work was funded by the National Natural Science Foundation of China (No. 61003065, and 61272169), the Specialized Research Fund for the Doctoral Program of Higher Education (No. 20101101120027), and the Excellent Young Scholars Research Fund of the Beijing Institute of Technology (No. 2010Y0711).

15. This feature was accomplished after the experiment presented in Section 4, and thus this feature was not involved in the experiment.

References

[1] T. Mens and T. Touwe, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126-139, Feb. 2004.
[2] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD dissertation, Univ. of Illinois at Urbana-Champaign, 1992.
[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison Wesley Professional, 1999.
[4] R. Arnold, "An Introduction to Software Restructuring," *Tutorial on Software Restructuring,* R. Arnold, ed. IEEE CS Press, 1986.
[5] W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 3, pp. 228-269, July 1993.
[6] M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.*, pp. 50:1-50:11, http://doi.acm.org/10.1145/2393596.2393655, 2012.
[7] E. Mealy and P. Strooper, "Evaluating Software Refactoring Tool Support," *Proc. Australian Software Eng. Conf.*, p. 10, Apr. 2006.
[8] Eclipse Foundation. Eclipse 3.4.2, http://www.eclipse.org/emft/projects/, 2013.
[9] R.M. Fuhrer, M. Keller, and A. Kiezun, "Advanced Refactoring in the Eclipse JDT: Past, Present, and Future," *Proc. First Workshop Refactoring Tools in Conjunction with 21st European Conf. Object-Oriented Programming,* D. Dig, ed., pp. 30-31, 2007.
[10] Microsoft Corporation, Microsoft Visual Studio 2008, http://www.microsoft.com/, 2013.
[11] JetBrains Company, IntelliJ IDEA 8, http://www.jetbrains.com/idea/, 2013.
[12] E. Mealy and P. Strooper, "Evaluating Software Refactoring Tool Support," *Proc. Australian Software Eng. Conf.*, pp. 331-340, 2006.
[13] W.C. Wake, *Refactoring Workbook.* Addison Wesley, Aug. 2003.
[14] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.*, vol. 28, no. 6, pp. 654-670, July 2002.
[15] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 347-367, May/June 2009.
[16] G. Bavota, A.D. Lucia, and R. Oliveto, "Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures," *J. Systems and Software,* vol. 84, no. 3, pp. 397-414, 2011.
[17] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y. Gueheneuc, "Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 1 -5, Sept. 2010.
[18] N. Tsantalis and A. Chatzigeorgiou, "Identification of Refactoring Opportunities Introducing Polymorphism," *J. Systems and Software,* vol. 83, no. 3, pp. 391-404, 2010.
[19] N. Moha, Y.-G. Guhneuc, and L. P., "Automatic Generation of Detection Algorithms for Design Defects," *Proc. 21st IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 297-300, Sept. 2006.
[20] N. Moha, Y.-G. Guhneuc, L. Duchien, and L.M. A.-F., "Decor: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20-36, Jan./Feb. 2010.
[21] T. Tourwe and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," *Proc. Seventh European Conf. Software Maintenance and Reeng.*, pp. 91-100, 2003.
[22] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test," *IEEE Trans. Software Eng.*, vol. 33, no. 12, pp. 800-817, Dec. 2007.
[23] M. Munro, "Product Metrics for Automatic Identification of 'Bad Smell' Design Problems in Java Source-Code," *Proc. 11th IEEE Int'l Symp. Software Metrics,* p. 15, Sept. 2005.
[24] E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 5-18, Jan./Feb. 2012.
[25] M.A. Cusumano and R.W. Selby, *Microsoft Secrets.* Free Press, 1995.
[26] B.W. Boehm, "Seven Basic Principles of Software Engineering," *J. Systems and Software,* vol. 3, no. 1, pp. 3-24, Mar. 1983.
[27] E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software,* vol. 25, no. 5, pp. 38-44, Sept./Oct. 2008.

[28] G. Travassos, F. Shull, M. Fredericks, and V.R. Basili, "Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality," *Proc. 14th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 47-56, http://doi.acm.org/10.1145/320384.320389, 1999.
[29] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. Second IEEE Working Conf. Reverse Eng.,* pp. 86-95, July 1995.
[30] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," *Proc. Seventh Working Conf. Reverse Eng.,* pp. 98-107, 2000.
[31] I. Baxter, A. Yahin, L. Moura, S. Anna, M., and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* pp. 368-377, 1998.
[32] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use During Preventative Maintenance," *Proc. Second IEEE Int'l Workshop Source Code Analysis and Manipulation,* pp. 36-43, Oct. 2002.
[33] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance,* pp. 109-118, 1999.
[34] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods," *J. Systems Software,* vol. 84, pp. 1757-1782, http://dx.doi.org/10.1016/j.jss.2011.05.016, Oct. 2011.
[35] R. Koschke, "Identifying and Removing Software Clones," *Software Evolution,* T. Mens and S. Demeyer, eds., pp. 15-36, Springer, 2008.
[36] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," *Proc. 13th Working Conf. Reverse Eng.,* pp. 253-262, 2006.
[37] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting Duplications in Sequence Diagrams Based on Suffix Trees," *Proc. 13th Asia Pacific Software Eng. Conf.,* pp. 269-276, 2006.
[38] H. Liu, W. Shao, L. Zhang, and Z. Ma., "Detecting Overlapping Use Cases," *IET Software (IEE Processing Software),* vol. 1, no. 1, pp. 29-36, Feb. 2007.
[39] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Trans. Software Eng.,* vol. 33, no. 9, pp. 577-591, Sept. 2007.
[40] C.K. Roy, J.R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," *Science of Computer Programming,* vol. 74, pp. 470-495, http://www.sciencedirect.com/science/article/pii/S0167642309000367, 2009.
[41] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: A Review of Current Knowledge," *J. Software Maintenance and Evolution: Research and Practice,* vol. 23, no. 3, pp. 179-202, http://dx.doi.org/10.1002/smr.521, 2011.
[42] N. Gode and R. Koschke, "Incremental Clone Detection," *Proc. European Conf. Software Maintenance and Reeng.,* pp. 219-228, 2009.
[43] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-Based Code Clone Detection: Incremental, Distributed, Scalable," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 1-9, Sept. 2010.
[44] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort," *IEEE Trans. Software Eng.,* vol. 38, no. 1, pp. 220-235, Jan./Feb. 2012.
[45] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Software Eng.,* vol. 33, no. 11, pp. 725-743, Nov. 2007.
[46] R. Robbes and M. Lanza, "A Change-Based Approach to Software Evolution," *Electronic Notes in Theoretical Computer Science,* vol. 166, pp. 93-109, 2007.
[47] R. Lincke, J. Lundberg, and W. Löwe, "Comparing Software Metrics Tools," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 131-142, http://doi.acm.org/10.1145/1390630.1390648, 2008.
[48] E. Murphy-Hill, A.P. Black, D. Dig, and C. Parnin, "Gathering Refactoring Data: A Comparison of Four Methods," *Proc. Second Workshop Refactoring Tools,* pp. 7:1-7:5., http://doi.acm.org/10.1145/1636642.1636649, 2008.
[49] B. Biegel and S. Diehl, "JCCD: A Flexible and Extensible API for Implementing Custom Code Clone Detectors," *Proc. 25th IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 167-168, Sept. 2010.
[50] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," *Proc. Ninth Working Conf. Reverse Eng.,* pp. 97-108, 2002.

Hui Liu received the BS degree in control science from Shandong University in 2001, the MS degree in computer science from Shanghai University in 2004, and the PhD degree in computer science from Peking University in 2008. He is an associate professor in the School of Computer Science and Technology at the Beijing Institute of Technology. He is particularly interested in software refactoring, design pattern, and software evolution. He is currently doing research to make software refactoring easier and safer. He is also interested in developing practical refactoring tools to assist software engineers.

Xue Guo received the BS degree in computer science from Shanxi Normal University in 2011, and is currently working toward the master's degree in the School of Computer Science and Technology at the Beijing Institute of Technology. She is interested in software refactoring and software evolution.

Weizhong Shao received the BS degree in mathematics and mechanics and the MS degree in computer science from Peking University in 1970 and 1983, respectively. He is a professor in the Institute of Software, School of Electronics Engineering and Computer Science at Peking University. His current research interests include software engineering, object-oriented technologies, software reuse, and component-based software development.