Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort

Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu

Abstract—Bad smells are signs of potential problems in code. Detecting and resolving bad smells, however, remain time-consuming for software engineers despite proposals on bad smell detection and refactoring tools. Numerous bad smells have been recognized, yet the sequences in which the detection and resolution of different kinds of bad smells are performed are rarely discussed because software engineers do not know how to optimize sequences or determine the benefits of an optimal sequence. To this end, we propose a detection and resolution sequence for different kinds of bad smells to simplify their detection and resolution. We highlight the necessity of managing bad smell resolution sequences with a motivating example, and recommend a suitable sequence for commonly occurring bad smells. We evaluate this recommendation on two nontrivial open source applications, and the evaluation results suggest that a significant reduction in effort ranging from 17.64 to 20 percent can be achieved when bad smells are detected and resolved using the proposed sequence.

Index Terms—Scheme, bad smell, software refactoring, effort, detection, schedule.

1 Introduction

1.1 Software Refactoring and Bad Smells

Software refactoring is used to restructure the internal structure of object-oriented software to improve its quality, especially its maintainability, extensibility, and reusability [1], [2], while preserving its external behavior. Software refactoring is widely used to delay the degradation effects of software aging and facilitate software maintenance. Because software is repeatedly modified according to evolving requirements, source code shifts from its original design structure. The source code becomes complex, difficult to read or debug, and even harder to extend. Software refactoring improves readability and extensibility by cleaning up bad smells in the source code.

A key issue in software refactoring is determining the kind of source code that requires refactoring. Experts have summarized typical situations that may require refactoring [3], [4]. Fowler et al. call them *Bad Smells* [3, Chapter 3], indicating that some part of the source code smells terrible. In other words, *Bad Smells* (e.g., *Duplicated Code*) are signs of potential problems in code that may require refactoring. The definition and explanation of bad smells can be found in the third chapter of their book. More bad smells can be found in other books [4] or on the Internet [5]. These bad smells are usually linked to corresponding refactoring rules that can help dispel these bad smells.

- *H. Liu is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China, and with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China, and also with the Beijing Lab of Intelligent Information Technology, School of Computer Science, Beijing Institute of Technology. E-mail: Liuhui08@bit.edu.cn.*
- *Z. Ma and W. Shao are with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China. E-mail: mzy@sei.pku.edu.cn, wzshao@pku.edu.cn.*
- *Z. Niu is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: zniu@bit.edu.cn.*

Manuscript received 16 May 2010; revised 16 Aug. 2010; accepted 25 Oct. 2010; published online 3 Jan. 2011.
Recommended for acceptance by K. Inoue.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-05-0149.
Digital Object Identifier no. 10.1109/TSE.2011.9.

1.2 Tool Support

Tool support is crucial for the success of bad smell detection and resolution [6], [7], [8] because of the following reasons.

First, uncovering bad smells in large systems necessitates the use of detection tools because manually uncovering these smells is tedious and time-consuming, especially those involving more than one file or package, e.g., *duplicated code*. The tools are expected to detect bad smells automatically or semiautomatically. Clone detection is an excellent example, and researchers have proposed detection algorithms [9], [10], [11], and developed tools [10], [12], [13], [14] for clone detection in the last decades.

Second, software engineers need tools to automatically or semiautomatically carry out refactorings to clean bad smells. Manual refactoring is time-consuming and error-prone. For example, renaming a variable requires revising all references to that variable. Manually identifying all references is challenging—an issue that detection tools based on program analysis seek to address. Most mainstream integrated development environments (IDE), such as Eclipse [15],[1] Microsoft Visual Studio [16], and IntelliJ IDEA [17], support software refactoring [18]. Professional refactoring tools have also been developed. Rich tool support, in turn, accelerates the popularization of software refactoring.

Human intervention, however, remains indispensable to bad smell detection and resolution [19] because of the following reasons.

First, most bad smells automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools.

Second, it is up to software engineers to determine how to restructure bad smells in terms of refactoring rules that should be applied, and arguments of the rules.

1. http://www.eclipse.org/.

Finally, not all refactorings are supported by refactoring tools. For example, Eclipse (version 3.6, created on June 8 2010) supports only 23 refactoring rules but the number of proposed refactoring rules has increased to 93.[2]

As a result, detecting and resolving bad smells remain time-consuming [20], even with tool support. Consequently, research aimed at simplifying detection and resolution is still urgently needed.

1.3 Detection and Resolution Sequence of Bad Smells

Software refactoring can be performed in two different ways. The first is XP-style small-step refactoring confined to a few local files. The other is a batch model in which a large system is thoroughly refactored in one attempt. The latter is the target scenario of this paper. A detailed discussion on refactoring scenarios is presented in Section 6.1.

In a batch model, different kinds of bad smells are typically detected and resolved individually. Suppose a software engineer, familiar with a list of bad smells and associated refactoring rules, refactors a large system. He is equipped with bad smell detection tools and automatic or semi-automatic refactoring tools for cleaning up bad smells. He first chooses a detection tool to identify a specific type of bad smell (a detection tool usually uncovers only one kind of bad smell, e.g., clone detection tools are insensitive to bad smells other than clones). The detection tool proposes initial results that require manual confirmation. Once the detected bad smell is confirmed, the software engineer decides how to refactor it. Selected refactoring rules are manually or semi-automatically applied to the bad smells with the help of refactoring tools. Then, the software engineer moves on to the next kind of bad smells, and repeats the process until all kinds of bad smells have been detected and resolved. As a result, different kinds of bad smells are detected and resolved one after the other (Fig. 1), regardless of whether the sequence is arranged consciously or unconsciously.

The scheme for bad smells is two-tiered (Fig. 1): kind-level and instance-level. The kind-level scheme arranges the detection and resolution sequences of different kinds of bad smells. For example, should *Large Class* smells be detected and resolved before the *Long Method* type? Instance-level scheme arranges resolution sequences for instances of a specific kind of bad smell (e.g., *Large Class*). For example, numerous clones may be detected in a large application, and an instance-level scheme should manage the sequence in which these clones are resolved. In this paper, we focus on the kind-level scheme.

Resolution of one kind of bad smells may influence (simplify or complicate) the detection and resolution of other bad smells. For example, *Duplicated Code* may cause *Long method*. Consequently, if *Duplicated Code* is resolved first, *Long method* caused by it may disappear as well. However, little is known about the impact of resolving one kind of bad smell on the detection and resolution of other (remaining) bad smells.

Different resolution sequences of the same set of bad smells may require different efforts because resolving one kind of bad smell may simplify or complicate the detection and resolution of others. Therefore, it is possible to simplify bad smell detection and resolution by arranging appropriate detection and resolution sequences. The sequence in which *Duplicated Code* and *Long method* smells are resolved is an excellent example. Resolving *Duplicated Code* before *Long Method* is easier than the reverse because *Long Method* may disappear as a result of resolving *Duplicated Code*. Likewise, it is possible to maximize the effect of software refactoring by resolving bad smells in an optimal sequence. The importance of resolution sequences is illustrated in detail in Section 2.

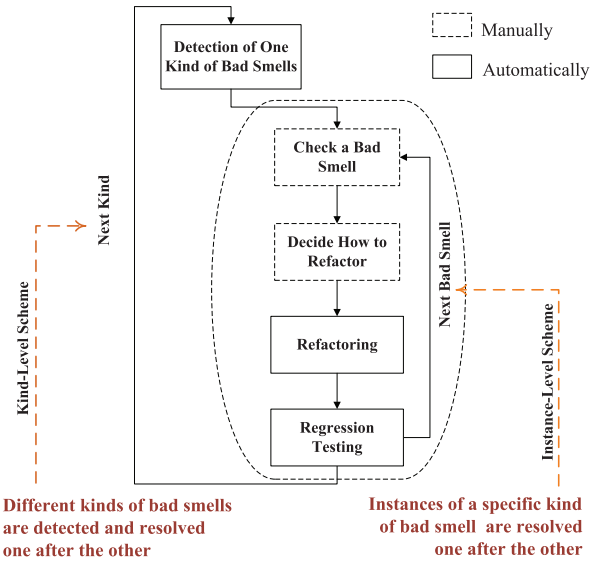


Fig. 1. Detection and resolution of bad smells.

To the best of our knowledge, thus far no published research focuses on the detection and resolution sequences of different kinds of bad smells (kind-level scheme in Fig. 1). Bouktif et al. [21] attempted to schedule bad smell resolution; however, their study focused on the resolution sequence of different instances of the same kind of bad smell (*Clone*). Their approach is an instance-level scheme depicted in Fig. 1.

This paper offers the following contributions: We analyze the relationships among different kinds of bad smells and their influence on detection and resolution sequences. We identify the need to arrange detection and resolution sequences of different kinds of bad smells using a motivating example. We also recommend a resolution sequence for commonly occurring bad smells, and validate it on two nontrivial applications. The experimental results suggest that a significant reduction in refactoring effort (ranging from 17.64 to 20 percent) can be achieved when bad smells are resolved using the recommended sequence.

The rest of the paper is structured as follows: Section 2 presents a motivating example that demonstrates the importance of resolution sequences of bad smells. In Section 3, we analyze the relations between different kinds of bad smells, and recommend a resolution sequence for these bad smells in Section 4. Section 5 presents the evaluation of the proposed scheme on two nontrivial applications. Section 6 discusses the practicability, informality, optimization, and extensibility of the proposed scheme. Section 7 presents a short overview of related research. The conclusions drawn are provided in Section 8.

2. Aug 2010, http://www.refactoring.com/catalog/index.html.

```
class Person {
 public:
   int m_telephoneNumber
}

class Teacher:Person{
 public:
  // Exposed Field + Inappropriate Name
   String m_Str;

  // Duplicate field in sibling subclasses
  int m_Age;

  Array<Borrow> borrowedBooks;
  public void PrintPersonalInfo()
   {
    cout<< "Name:" << m_Str;
    cout<<"Age:" <<m_Age;
    cout<<"Tel:"  << m_telephoneNumber
   }

 // Long Method
 public void PrintAll(){
 // Duplicated Code
   cout<< "Name:" << m_Str;
   cout<<"Age:" <<m_Age;
   cout<<"Tel:"  << m_telephoneNumber
 // End of Duplicated Code

   foreach (Borrow borrow in borrowedBook)
     {
      // Feature Envy
      cout<<"Book Name:"  <<borrow.book.m_Name;
      cout<<"Borrow Date:" <<borrow.m_BorrowDate;
      cout<<"Return Date" <<borrow.m_ReturnDate;
      cout<<"Is Charged:" <<borrow.m_IsCharged;
      cout<<"Is Renewed"  <<borrow.m_IsRenewed;
     // End of Feature Envy
    }
}

class Student:Person {
  public:
   Major m_Major;
   // Duplicate field in sibling subclasses
   // Useless field
   int m_Age;

   public void Print()
     {
      cout<<"Major:" <<m_Major;
      cout<<"Tel:"  << m_telephoneNumber
     }
}
```

Fig. 2. Motivating example.

2 Motivating Example

2.1 Sample Source Code and Bad Smells

To demonstrate the importance of resolution sequences of bad smells, we present a motivating example, depicted in Fig. 2. The source code was extracted from a library management system, and slightly modified for demonstration. In the sample source code (in C++), we point out a number of bad smells along with comments.

2.2 Exposed Field versus Inappropriate Name

The first bad smell comes from public field *m_Str* of class *Teacher*. First, we do not encourage developers to expose fields of classes because this may reduce the modularity and encapsulation of the program. Instead, we encourage them to encapsulate these fields with refactoring rule *Encapsulate Field* [3]. Second, the name of the field does not reveal its intent. Naming the field *m_Name* is more advisable.[3] The

3. It is debatable to name variables with the prefix "*m_*". The organization developing the library management system insists on this style (similarly to what Microsoft did in Microsoft Foundation Class Library, MFC); thus we do not treat it as a bad smell.

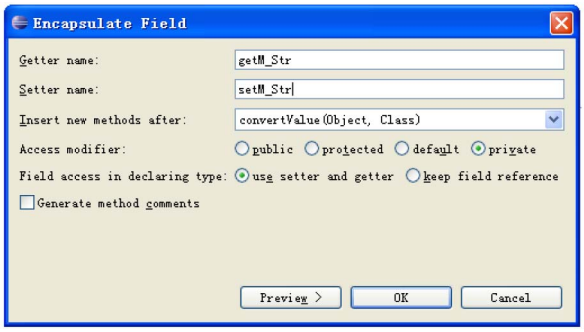


Fig. 3. Encapsulate field.

corresponding refactoring rule for removing this bad smell is *Rename Field*.

Two kinds of bad smells (*Exposed Field* and *Inappropriate Name*) are found on element *m_Str*. They require different refactorings according to different refactoring rules (*Encapsulate Field* and *Rename Field*, respectively). Eclipse cannot simultaneously address them; they have to be dealt with individually.

Suppose we decide to resolve *Exposed Field* with Eclipse first, using refactoring rule *Encapsulate Field*. We select the field, right click and select menu-item *Refactor* and its submenu-item *Encapsulate Field*. Then, the refactoring tool proposes a diagram for us to specify method names, as shown in Fig. 3. The default method names are *getM_Str* and *setM_Str*. The coding style forbids the infix "*M_*" in method names. Thus, we remove "*M_*" from the default method names, click *Ok*, and the refactoring tool generates the following result:

```
private:
   String m_Str;
public:
 void setStr(String m_Str)
      {
        this.m_Str = m_Str;
      }
 String getStr()
     {
        return m_Str;
     }
```

After the resolution of *Exposed Field*, we proceed to other tasks. After quite a long time, we come back to field *m_Str* to resolve the other bad smell *Inappropriate Name* using refactoring rule *Rename Field*. The work is also supported by Eclipse. We select field *m_Str*, right click it, and select menu-item *Refactor* and its submenu-item *Rename*. Then, the refactoring tool proposes a diagram for us to specify the new name as shown in Fig. 4.

We change the name to *m_Name*, and Eclipse automatically modifies the source code as follows:

```
private:
   String m_Name;
public:
 void setStr(String m_Str)
         {
           this.m_Name = m_Str;
         }
 String getStr()
         {
           return m_Name;
         }
```

Although Eclipse has changed all occurrences of the field, it is far from perfect. The names of *getter* and *setter* methods

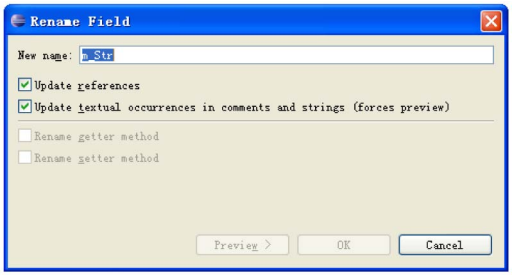


Fig. 4. Rename field.

are not modified accordingly, and the parameters of the methods are not changed, either. For clients of the class, the names of the methods and parameters are even more critical than that of the encapsulated field. However, automatically detecting whether a method is the *getter* or *setter* method of a specific field is difficult. First, names of *getter* and *setter* methods may vary according to different coding styles, which makes it difficult to judge using method names as bases. Second, the computation in the *getter* and *setter* methods (especially in *setter*) may be complicated (computation to ensure the field is correctly assigned), which makes it challenging, if not impossible, to evaluate using the computation within the methods as bases.

One way to overcome this problem is to reverse the resolution sequence of the two bad smells: resolve *Inappropriate Name* before *Exposed Field*. We rename the field first, and then encapsulate the renamed field. The *getter* and *setter* methods will then be named properly, as well as their parameters.

2.3 Duplicated Field in Sibling Subclasses versus Dead Field

The next bad smell is *Duplicated Field in Sibling Subclasses*: field *m_Age* appears in both subclasses of *Person*. According to [3], duplicated fields in sibling subclasses should be pulled up to their superclass by refactoring rule *Pull up Field*.

We find another kind of bad smell from field *m_Age*; the field defined in class *Student* has never been used. In other words, it is useless and should be removed.

We find two kinds of bad smells on the same element. Will the resolution sequence influence refactoring results as in the previous example?

Suppose we resolve *Duplicated Field in Sibling Subclasses* first, and field *m_Age* is pulled up to superclass *Person*. As a result, bad smell *Dead Field* disappears as well as *Duplicated Field in Sibling Subclasses*. This, however, yields another kind of bad smell: *Refused Bequest* [3]. According to the rule of inheritance, *Student* should inherit field *m_Age* from its superclass, but it refuses to acquire it because it has no use for that. Consequently, we have to refactor it again, and pull down field *m_Age* from the superclass to subclass *Teacher*. Pulling up a field, then immediately pulling it down is not only time-consuming, but also error-prone. The more frequently it is performed, the higher the possibility of mistakes. It is also likely to miss the latter refactoring (pull down the field), which influences the refactoring results.

Resolving *Dead Field* before *Duplicated Field in Sibling Subclasses* would be easier. Once *Dead Field* is removed from *Student*, *Duplicated Field in Sibling Subclasses* disappears and no further refactoring is required.

2.4 Duplicated Code and Feature Envy versus Long Method

The last group of bad smells is composed of *Duplicated Code*, *Feature Envy*, and *Long Method*, emitted from method *PrintAll* of class *Teacher*.

The first part of method *PrintAll* is the same as that in method *PrintPersonalInfo*. We remove the duplication by replacing the instructions with a single method call. The justification for this approach can be found in [3].

The computation within the *foreach* statement refers to many fields of class *Borrow*, which is a typical *Feature Envy*. We extract the fragment as a new method, move it to class *Borrow*, and then insert a method call in the original location.

Because of space constraints, presenting a method long and complex enough to be called *Long Method* is difficult. To simplify the demonstration, we treat methods with more than seven lines of source code as long methods. Thus, method *PrintAll* is a long method.

If we resolve *Duplicated Code* and *Feature Envy* first, we may find that once *Duplicated Code* and *Feature Envy* are cleaned, *Long Method* disappears automatically: the length of the method is reduced dramatically. *Duplicated Code* and *Feature Envy* are detected by tools, such as CCFinder [10] (for *Duplicated Code*) and JDeodorant [22] (for *Feature Envy*), which makes the refactoring activities more convenient.

However, if *Long Method* is resolved first, things become more complicated. We have to manually decompose the long method [3]. Manual decomposition is a lengthy and error-prone process. By analyzing the method, we may uncover the duplicated code because the two duplicated fragments are physically close to each other, and reduce the length by resolving the duplicated code. But if the involved methods are positioned far from each other, detecting duplications while resolving *Long Method* would be difficult, if not impossible.

2.5 Summary

The motivating example demonstrates the importance of the resolution sequences of bad smells. Inappropriate resolution sequences may require more effort, as indicated in Sections 2.3 and 2.4. Furthermore, inappropriate resolution sequences may also reduce the effect of refactoring, as indicated in Section 2.2.

3 Analysis

Bad smells have not yet been completely formalized. To date, bad smells are usually described in an informal or semiformal way [3], [4], [23]. That is the reason why the detection and resolution of bad smells are not completely automated (Fig. 1 indicates where human intervention is needed). This situation makes it difficult to find a formal method for analyzing the effect of resolving one kind of bad smell on detecting and resolving other kinds of bad smells. Consequently, these effects are analyzed in an informal way. The detailed discussion on why formal analysis is currently impractical is presented in Section 6.2.

TABLE 1
Evaluated Bad Smells

	Bad Smells
1	Duplicated Code
2	Long Method
3	Large Class
4	Long Parameter List
5	Feature Envy
6	Primitive Obsession 6.1: Simple Primitive Obsession 6.2: Simple Type Code 6.3: Complex Type Code
7	Useless Field
8	Useless Method
9	Useless Class

3.1 Overview

To find out potential relations between different bad smells, we ask the following questions for each pair of bad smells:

- Will different resolution sequences generate different resulting systems? If so, why?
- Will the resolution of a bad smell make the detection of the other more convenient? If so, why?
- Will the resolution of a bad smell complicate the detection of the other? If so, why?
- Will the resolution of a bad smell simplify the process of modifying (resolving) the other bad smell? If so, why? Modification here includes all operations on detected bad smells.
- Will the resolution of a bad smell complicate the modification of the other? If so, why?

It is also possible that resolution sequences do not matter for a given pair of bad smells. For example, it does not matter whether *Simple Primitive Obsession* is resolved before *Feature Envy*. For this case, we leave them alone, and no resolution sequence is recommended.

3.2 Evaluated Bad Smells

As an initial study, we consider nine kinds of bad smells for evaluation. The evaluated bad smells are listed in Table 1. A detailed explanation of these bad smells can be found in the book by Fowler et al. [3, Chapter 3]. A brief introduction is presented here so that the paper can be understood on its own.

- **Duplicated Code.** *Duplicated Code* is comprised of fragments of source code appearing in more than one location. In a narrow sense, only identical copies are called *Duplicated Codes*. In a broad sense, however, they may include slightly different fragments resulting from *copy-paste-modify* editions.
- **Long Method.** The longer a method is, the harder it is to read or modify. Consequently, long and complex methods should be divided into short and well-named methods with refactoring rules, e.g., *Extract Method*.
- **Large Class.** Large classes usually try to take too many responsibilities, making them complex and confusing. To improve their readability and maintainability, large classes should be divided into smaller ones, each for a single responsibility.
- **Long Parameter List.** Methods with too many parameters are difficult to use and even harder to change. The parameters can ordinarily be replaced with a few objects that contain the parameters.
- **Feature Envy.** A method or a fragment of a method may be more interested in features of another class than those of the enclosing class, which is called *feature envy*.
- **Primitive Obsession.** *Primitive Obsession* is the situation in which objects should have been used instead of primitives. It is further divided into three subcategories: *Simple Primitive Obsession*, *Simple Type Code*, and *Complex Type Code* [3]. The division is necessary because different refactoring rules should be applied depending on whether the primitive object is a type code and whether the type code influences the behavior of the enclosing class. If the primitive object is not a type code, the refactoring is simple: The data value is replaced with an object. If it is a type code, the refactoring is more complicated, depending on whether the type code affects behavior. If it does, refactoring rule *Replace Type Code with Subclasses* should be applied. Otherwise, *Replace Type Code with Class* is preferred.
- **Useless Field.** It is a synonym of *Dead Field*, referring to fields defined but never used.
- **Useless Method.** Once a domain class is found, designers may propose methods (operations) for it. Unfortunately, sometimes certain operations are irrelevant to the role the class plays in the specific system. These operations are useless in the system.
- **Useless Class.** Useless classes are those defined but never used. Typically, these are results of inappropriate boundaries of systems.

3.3 Pairwise Analysis

3.3.1 Useless Class versus Others

The removal of a useless class would simultaneously remove all bad smells associated with that class. Therefore, *Useless Class* should be removed before other bad smells.

3.3.2 Useless Method versus Others

The removal of a useless method would simultaneously remove all bad smells associated with that method. Thus, *Useless Method* should be removed before other bad smells, such as *Duplicated Code*, *Long Parameter List*, *Feature Envy*, and *Primitive Obsession*, that may appear in methods.

Large Class should be resolved after *Useless Method* because removing useless methods might reduce the size and complexity of large classes, making for easier resolution of *Large Class*.

Useless Method should be resolved before *Useless Field* because removing useless methods simplifies the detection of useless fields. To determine whether a field is useless, software engineers usually have to determine whether it is accessed by useful methods. However, once *Useless Method* is resolved, all remaining methods are useful. Consequently, detecting useless fields is simplified: Fields not

accessed by any method are simply detected. It is meaningful because determining whether a method is useless is time-consuming and error-prone. Existing IDEs, such as Eclipse, report only useless private methods. Other useless methods, public or protected, are difficult to detect, and thus have not yet been reported.

The only bad smell that should be resolved before *Useless Method* is *Useless Class*. The removal of a useless class removes all useless methods defined in that class.

3.3.3 Useless Field versus Others

Useless Field should be resolved after *Useless Class* and *Useless Method* as analyzed in preceding sections. There is no obvious interaction between *Useless Field* and other bad smells (other than *Useless Class* and *Useless Method*). There is little likelihood for *Useless Field* to overlap with other bad smells except for *Useless Class* and *Useless Method*.

3.3.4 Duplicated Code versus Long Method and Large Class

It is highly likely that long methods (and large classes) contain duplicated code. In this case, removing *Duplicated Code* shortens its enclosing long methods and large classes, simplifying the resolution of *Long Method* and *Large Class*. The reasons for this are discussed in the motivating example in Section 2.4.

3.3.5 Duplicated Code versus Primitive Obsession, Long Parameter List, and Feature Envy

The resolution of *Duplicated Code* replaces duplicated fragments with a single fragment and a few method calls, reducing the occurrences of bad smells within the fragments. Therefore, *Duplicated Code* should be resolved before those that may appear in duplicate fragments (e.g., *Primitive Obsession*, *Long Parameter List*, *Feature Envy*, *Simple Type Code*, and *Complex Type Code*).

Another reason why *Duplicated Code* should be resolved before *Simple Primitive Obsession* is that *Replace Primitive Data with Objects* as a solution to *Simple Primitive Obsession* would modify duplicated code. It is likely that not all duplicated fragments are modified in the same manner, which would complicate the detection of duplications.

3.3.6 Long Method versus Large Class

To eliminate *Large Class*, large classes are decomposed into a few small ones. On the other hand, long methods are decomposed into a few small methods to dispel *Long Method*. Consequently, resolving *Large Class* and *Long Method* leads to redistribution of responsibilities at class and method levels, respectively. Carrying out the redistribution of responsibilities from the bottom up makes for reasonable and thorough redistribution.

3.3.7 Long Method versus Long Parameter List

As a solution to *Long Method*, some parts of the method may be extracted as new methods. Usually the extracted new methods are called within the old one in the original location; thus, the extraction does not shorten the parameter list.

Different results may be generated if the extracted and remaining parts of the method play two parallel responsibilities. In this case, the responsibility of the old method may be redefined, excluding that of the extracted part. As a result, the parameters used by the extracted part are removed from the original method, reducing the length of its parameter list. Therefore, *Long Method* should be detected and resolved before *Long Parameter List*.

3.3.8 Long Method versus Feature Envy

During the resolution of *Feature Envy*, some parts of the involved method may be extracted and moved to another class. This is a special kind of method decomposition. The decomposition would reduce the length and complexity of the method and simplify the resolution of (or dispel) *Long Method* if the method happens to be a long one.

3.3.9 Long Method versus Primitive Obsession

Complex Type Code, a subtype of *Primitive Obsession*, usually results in long and complex switch statements. These complex switch statements are likely to cause *Long Methods*. In this case, the resolution of *Complex Type Code* would remove *Long Method*.

The resolution of *Simple Primitive Obsession*, another subtype of *Primitive Obsession*, would extract the primitive field and operations on it as an object. It may reduce the complexity of long methods containing operations on the primitive field and thus simplify the decomposition.

No obvious interaction between *Simple Type Code*, the third subtype of *Primitive Obsession*, and *Long Method* is observed. *Simple Type Code* is addressed by extracting enumerate fields (type code) as a reusable type, e.g., *enum* in C++. This has little impact on long methods. Thus, *Simple Type Code* and *Long Method* can be detected and resolved in any order.

3.3.10 Large Class versus Long Parameter List

Long Parameter List should be resolved after *Large Class* because the resolution of *Long Parameter List* would complicate the resolution of *Large Class*. A detailed explanation is presented as follows:

If a parameter list contains many items (fields) of a specific class, the items had better be replaced with an instance of the class to reduce the number of parameters (as a solution to *Long Parameter List*). After that, the class referred to in the new parameter list is decomposed for reasons such as *Large Class*. As a result, all instances of the class are replaced with instances of new classes introduced in the decomposition. The instance in the parameter list introduced while resolving *Long Parameter List* should be replaced as well. But the replacement is quite difficult because the implementation of the method should be checked carefully to determine how to replace the parameter.

3.3.11 Large Class versus Feature Envy

The resolution of *Feature Envy* involves decomposing classes. To remove *Feature Envy*, some part of the enclosing class is extracted and moved to another class. As a result, the complexity of the enclosing class is reduced. This will simplify the resolution of (or even dispel) *Large Class* if the enclosing class happens to be a large one.

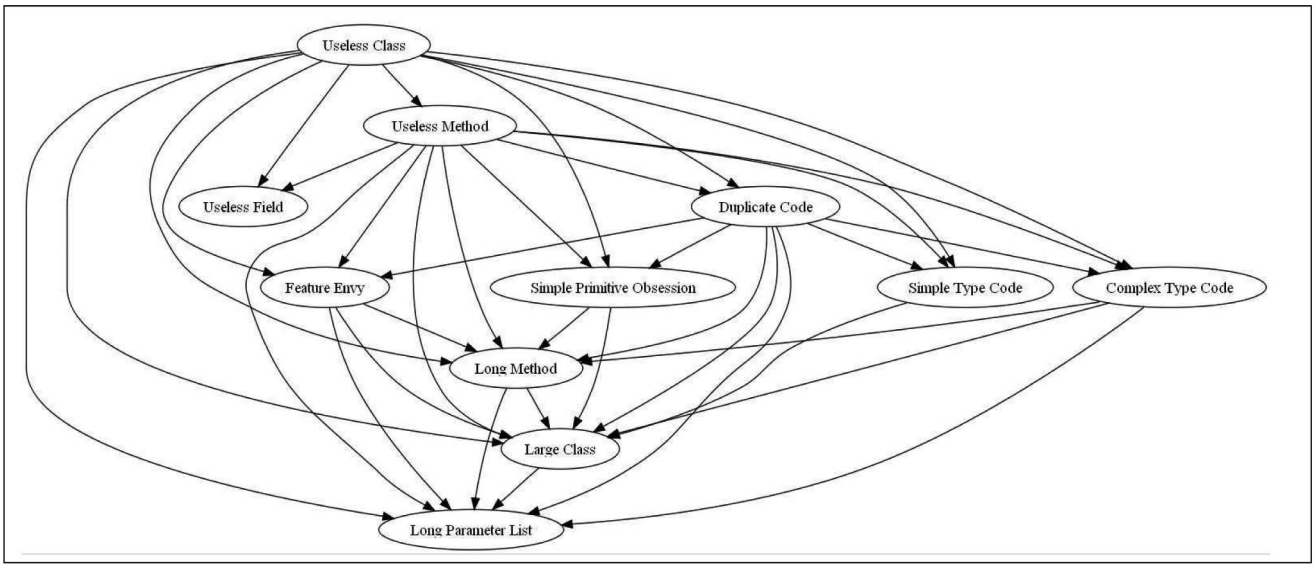


Fig. 5. Original pairwise resolution sequences of bad smells.

3.3.12 Large Class versus Primitive Obsession

While addressing *Primitive Obsession*, the enclosing class would be decomposed. Consequently, the size and complexity of the enclosing class would be reduced. If the enclosing class happens to be a large class, the reduction in size and complexity would simplify the removal of *Large Class*. Thus, *Primitive Obsession* had better be resolved before *Large Class*.

3.3.13 Long Parameter List versus Feature Envy

If a method is moved to the envied class as a solution to *Feature Envy*, some parameters of the method may be removed because they can be directly accessed within its enclosing class. This reduces the parameter list. Hence, *Feature Envy* had better be detected and resolved before *Long Parameter List*.

3.3.14 Long Parameter List versus Primitive Obsession

To resolve *Complex Type Code*, the enclosing class is decomposed. However, resolving *Long Parameter List* would complicate class decomposition, as discussed in Section 3.3.10. In other words, the resolution of *Long Parameter List* would complicate the resolution of *Complex Type Code*. Thus, *Long Parameter List* had better be detected and resolved after *Complex Type Code*.

The resolution of *Simple Primitive Obsession* and *Simple Type Code* would not change the fields of the enclosing class, although this may also result in class decomposition. Thus, this would not affect the resolution of *Long Parameter List*. As a result, they can be detected and resolved in any order.

3.3.15 Feature Envy and Primitive Obsession

No clear interaction between the resolution of *Feature Envy* and *Primitive Obsession* is observed. Hence, these can be resolved in any order.

4 Resolution Sequence of Bad Smells

With the analysis results in Section 3, this section tries to recommend a resolution sequence for the evaluated bad smells. The algorithm is based on the following assumption: All bad smells of the same kind are scheduled as a single unit. In other words, all bad smells of the same kind would be detected and resolved before the next kind of bad smell is detected (as depicted in Fig. 1). This assumption is based on the process of bad smell detection and resolution discussed in Section 1.3. As analyzed in that section, the assumption holds if refactorings are carried out in batch model.

First, we represent the analysis results of Section 3 with a directed graph (Fig. 5). Each vertex of the directed graph represents a category of bad smells whose name is presented in the label of the vertex. Each directed edge represents a preferred resolution sequence of the two kinds of bad smells represented by adjacent vertices of the edge, i.e., edge $\langle v_1, v_2 \rangle$ indicates that v_1 had better be resolved before v_2.

Although the resulting graph is exhaustive, it is overly complicated to be presented to software engineers as recommended guidance. Consequently, we simplify the graph by removing redundancies in the next step. We apply an algorithm to the directed graph to obtain a clear resolution sequence of different kinds of bad smells. One of the simplest ways to achieve the final resolution sequence is through *topological sorting*. However, when two or more vertices are available (whose in-degree is zero) in each iteration, topological sorting algorithms would randomly pick one of them for the next execution step. But we hope to leave the choice to software engineers who may take other factors into consideration, such as the severity of bad smells. Consequently, we propose a new algorithm for the final resolution sequences: removing redundant edges from Fig. 5.

For convenience, we define the following symbols:

- $p(v_1, v_2)$: A path from vertex v_1 to vertex v_2 containing more than one edge.
- $e(v_1, v_3)$: A direct edge from vertex v_1 to vertex v_3.

An edge $e(v_1, v_3)$ is redundant if and only if there is another path $p(v_1, v_3)$ in parallel to $e(v_1, v_3)$. The graph in Fig. 6 is a good example. Removing edge $e(v_1, v_3)$ would not change the topological order of the vertices, but removing any other edge from Fig. 6 would result in different topological order.

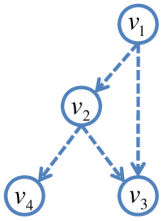


Fig. 6. Redundant edge.

The algorithm removing redundant edges is presented in Fig. 7. If the algorithm is applied to the graph in Fig. 6, the algorithm would remove $e(v_1, v_3)$ because the path $p(v_1, v_2, v_3)$ is parallel to $e(v_1, v_3)$. Other edges would be retained because no path is longer than 1 in parallel to $e(v_1, v_2)$, $e(v_2, v_3)$, or $e(v_2, v_4)$. The output of the algorithm on Fig. 7 is presented in Fig. 8.

The complexity of the algorithm is $O(|e|^2)$: $O(|e|)$ for the two loops on lines 3 and 5, another $O(|e|)$ for checking parallel paths on line 7. The complexity is not a huge problem. First, the graph is small and the algorithm can complete the computation quickly. On the nine kinds of bad smells analyzed in this paper (yielding a small graph with 38 edges), the algorithm completes the computation within 1 second (Window XP, Intel Core2 3.0 GHz, 2 GB memory). Second, the algorithm is run only once. Software engineers do not call the algorithm every time an application is refactored. Only when additional kinds of bad smells aside from those listed in Section 3.2 are introduced will the algorithm be called.

The algorithm is deterministic, i.e., the same result is produced on the same input graph regardless of the order in which the edges and vertices are checked. The algorithm produces nothing but the longest paths between every pair of reachable vertices. For the graph in Fig. 6, the final result is a set of the longest paths between every pair of reachable vertices, i.e., $<v_1, v_2>, <v_1, v_2, v_3>, <v_1, v_2, v_4>, <v_2, v_4>, <v_2, v_3>$.

```
1  /*Input:  Directed graph with redundancies
2    Output: Directed graph without redundancies */
3  for each vertex v in the graph
4    {
5     for each edge e(v,d) of vertex v
6      {
7        if there is another path p(v,d) besides e(v,d)
8           {
9             //the length of p(v,d) must be longer than
10            //that of e(v,d) because there are no
11            //parallel edges in the graph
12
13            remove e(v,d)
14
15            //Removal of e(v,d) would not change
16            //the topological orders
17          }
18     }
19   }
```

Fig. 7. Algorithm for removing redundancies.

5 Evaluation

In this section, we present an evaluation of the proposed approach.

5.1 Research Questions

We investigate the following research questions:

- Do bad smells overlap in the manner analyzed in Section 3? If so, how often?
- Does the detection and resolution sequence of bad smells recommended in this paper simplify software refactoring? If so, to what extent?

5.2 Subject Applications

The evaluation was performed on two open source applications downloaded from *SourceForge*: *Java Source Metric*[4] and *Thout Reader*.[5]

Java Source Metric was designed to measure the metrics of Java source code. It is an Eclipse plug-in. For any Java project opened in Eclipse, *Java Source Metric* would calculate its metrics. It is also possible to calculate metrics for a single source code unit, e.g., a class or a method. The latest version of *Java Source Metric* is 1.4.2 released in January 2010. Its source code is publicly available at *SourceForge* and *Trustie* (trustworthy software tools and integration environment).[6] It is composed of nearly 3,695 lines of source code.

Thout Reader is a reader allowing users to browse, search, bookmark, and append documents. The stable version of *Thout Reader* is 1.9.2 released on 11 December 2005. But the source of version 1.9.2 is not publicly available; thus we download the source code of version 1.8.0 instead, the latest version whose source code is published. It is composed of nearly 23,000 lines of source code (in Java).

Detailed information of the applications are presented and compared in Table 2. From the table, we observe that the applications come from different domains, and were developed by different developers. Carrying out evaluation on these applications may help generalize the conclusions.

5.3 Process

To analyze the effect of the recommended resolution sequence, we needed two teams to independently resolve bad smells in two identical copies of the evaluation subjects. Each team was composed of four graduates of computer science, and each of the graduates had taken part in refactoring of at least one nontrivial system before the evaluation.

First, we asked the teams to detect and resolve bad smells before they were informed about the resolution sequence of bad smells; they were allowed to detect and remove bad smells in any order they preferred. In this phase, Team A detected and resolved bad smells in *Java Source Metric*, while Team B detected and resolved bad smells in *Thout Reader*.

In the second phase, we briefed the teams on the idea proposed in this paper, and they were asked to detect and resolve bad smells in the recommended sequence in Fig. 8. In this phase, the two teams exchanged their evaluation applications: Team A detected and resolved bad smells in *Thout Reader* whereas team B detected and resolved bad smells in *Java Source Metric*. The exchange is valuable in

4. http://sourceforge.net/projects/jsourcemetric/.
5. http://thout.sourceforge.net/.
6. http://www.trustie.net/projects/files/list/PKUCodeMetric.

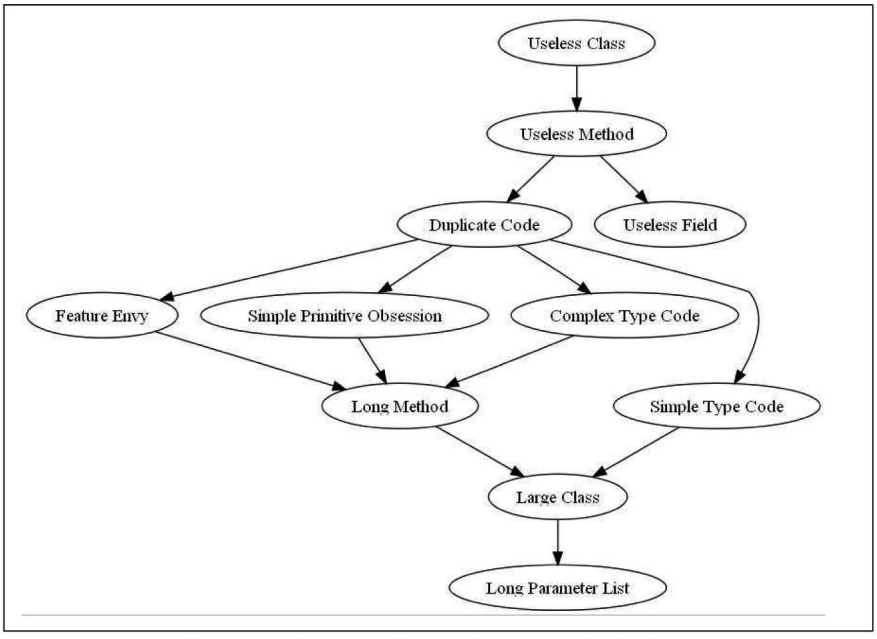


Fig. 8. Refined resolution sequences of bad smells.

ensuring the validity of the evaluation. Detailed reasons are discussed in Section 5.6.

Both teams detected bad smells and carried out refactorings with semiautomatic tool support. Bad smells were detected by the following tools (manual checking and detecting remain indispensable even with the help of these tools).

- *Duplicate Code* was detected by *PMD*[7] with $Minimal\ Duplicate\ Chunk = 50$.
- *Long Method*, *Large Class*, and *Long Parameter List* were detected by *Metrics*.[8] Thresholds for the bad smells were

  $(LOC > 50 \vee McCabeCyclomaticComplexity > 10)$,
  $(LOC > 100 \vee McCabeCyclomaticComplexity > 20)$,
  $(Parameter > 4)$, respectively.

- *Feature Envy* was detected by *JDeodorant*[9] [22].
- *Primitive Obsession*, *Useless Field*, *Useless Method*, and *Useless Class* were detected by the tools developed by the authors of this paper.

Once the refactoring was completed, the efforts of the teams,and the quality of the resulting systems were compared to investigate whether the recommended resolution sequence helped simplify bad smell detection and resolution. The measured efforts include those spent on testing after refactoring to make sure the applications worked correctly.

5.4 Results

Evaluation results are presented in Table 3. Row 4 shows how many times these bad smells overlapped. We counted only the overlaps between different kinds of bad smells; overlaps between bad smells belonging to the same kind were excluded because we were evaluating the detection and resolution sequences of different kinds of bad smells (kind-level scheme in Fig. 1). The instance-level scheme in Fig. 1 is beyond the scope of this paper. Row 9 shows the efforts spent on removing bad smells.

In *Java Source Metric*, 93 bad smells were found, including 65 duplications, 17 large classes, 3 long methods, and 8 feature envies. These bad smells were distributed in 27 files. Overlaps between these bad smells are presented in Tables 4 and 5. Table 4 presents the frequencies of overlaps between different kinds of bad smells. Table 5 presents the statistics collected for each kind of bad smells.

To detect and resolve these bad smells, Teams *A* and *B* worked on *Java Source Metric* independently. It took *Team A* (nonscheduled) 10 man-days, whereas it only took *Team B* (scheduled) 8 man-days. In other words, $(10 - 8)/10 = 20\%$ efforts were conserved by the scheme.

In *Thout Reader*, 232 bad smells were found, including: 152 duplications, 10 large classes, 42 long methods, 7 long

TABLE 2
Subject Applications

	Java Source Metric	Thout Reader
Programming Language	Java	Java
Developer	Chao Guo, Liu Shi	Mark Carey, Gary Varnell, Rick Blair
Size (LOC)	3,695	23,000
Number of Classes	60	270
Number of Methods	268	1147
Version	1.4.2	1.8.0
Release Date	2010	2005
Domain	Metrics	Reader

7. http://pmd.sourceforge.net/cpd.html.
8. http://metrics.sourceforge.net/.
9. http://www.jdeodorant.com/.

TABLE 3
Evaluation Results

	Java Source Metric		Thout Reader	
	Non-Scheduled	Scheduled	Non-Scheduled	Scheduled
Number of Bad Smells	93		232	
Overlaps of Bad Smells	45		145	
Bad Smells Overlapped	53 (53/93 = 56.99%)		113 (113/232 = 48.71%)	
Code Units Containing Bad Smells	27		48	
Code Units Containing Overlapping Smells	12 (12/27 = 44.44%)		11 (11/48 = 22.92%)	
Experiment Participator	Team A	Team B	Team B	Team A
Effort (Man-Day)	10	8	17	14
Reduction in Effort	—	(10-8)/10=20%	—	(17-14)/17=17.65%

TABLE 4
Overlaps in *Java Source Metrics*

Smell # 1	Smell # 2	Number of Overlaps
Duplicate Code	Large Class	34
Feature Envy	Large Class	6
Duplicate Code	Feature Envy	2
Large Class	Long Method	3
Total		45

TABLE 5
Statistics of Overlaps in *Java Source Metrics*

Bad Smells	Total	Overlapped	Ratio
Duplicate Code	65	30	30/65=46.15%
Large Class	17	14	14/17=82.35%
Long Method	3	3	3/3=100%
Feature Envy	8	6	6/8=75%
Total	93	53	53/93=56.99%

TABLE 6
Overlaps in *Thout Reader*

Smell # 1	Smell # 2	Number of Overlaps
Duplicate Code	Long Parameter List	4
Duplicate Code	Large Class	52
Long Method	Long Parameter List	2
Feature Envy	Large Class	12
Duplicate Code	Feature Envy	2
Large Class	Long Method	22
Duplicate Code	Long Method	50
Feature Envy	Long Method	1
Total		145

TABLE 7
Statistics of Overlaps in *Thout Reader*

Bad Smells	Total	Overlapped	Ratio
Duplicate Code	152	66	66/152=30.26%
Large Class	10	6	6/10=60%
Long Method	42	27	27/42=64.29%
Feature Envy	21	12	12/21=57.14%
Long Parameter List	7	2	12/21=57.14%
Total	232	113	113/232=48.71%

parameter lists, and 21 feature envies. The bad smells were distributed in 48 files. Overlaps between these bad smells are presented in Tables 6 and 7.

Detecting and resolving these bad smells took *Team B* (non-scheduled) 17 man-days, whereas it only took *Team A* 14 man-days. In other words, $(17 - 14)/17 = 17.65\%$ efforts were conserved using the proposed scheme.

5.5 Answers to Research Questions

From the evaluation results, we make the following observations:

- First, different kinds of bad smells overlap with each other. As suggested by the evaluation results in Table 3, nearly 50 percent of bad smells overlapped in the manner predicted in Section 3.
- Second, the bad smell detection and resolution scheme helps save effort. In *Java Source Metric* and *Thout Reader*, the scheme saved 20 and 17.65 percent of the efforts exerted, respectively.

5.6 Threats to Validity

A threat to internal validity is the employment of different experimenters in scheduled and nonscheduled detection and resolution of bad smells. The differences in the experimenters might influence detection and refactoring efforts. Scheduled and nonscheduled bad smell resolution were carried out by different teams; thus, the evaluation results can be influenced by the difference in teams instead of the resolution sequences. To reduce this threat, we exchanged the roles of the teams during the evaluation. Team A carried out scheduled and nonscheduled bad smell resolution on *Thout Reader* and *Java Source Metric*, respectively, whereas Team B carried out scheduled and nonscheduled bad smell resolution on *Java Source Metric* and *Thout Reader*, respectively. In other words, Teams A and B exchanged their roles (nonscheduled or scheduled) when the evaluated application was switched from *Thout Reader* to *Java Source Metric*. However, despite the exchange in roles, the evaluation on the two applications yielded the same conclusion: The recommended detection and resolution sequence entailed less effort on bad smell detection and resolution.

Another threat to internal validity is that if experimenters had known the expectation of the evaluation in advance,

experimenters of scheduled refactoring might work harder, whereas those of nonscheduled refactoring might not exert as much effort. To moderate this threat, we isolated the teams from each other and they were not informed that they were carrying out an evaluation. Instead, we told them that an improved version of the subjects was urgently needed.

A threat to external validity is the subjects of the evaluation. Certain specific characteristics (such as specific distribution of bad smells) of the subjects may affect our evaluation results. To reduce this threat, we carried out the evaluation on two applications from different domains developed by different developers. This approach helped generalize the conclusions.

Another threat to external validity is that experimenters in the evaluation were graduates majoring in computer science. Although all of them had rich experience in software development and refactoring, the conclusions derived from graduates are not necessarily true for experienced software engineers.

A threat to construct validity is that the detection tools employed in the evaluation might miss some bad smells. Missed bad smells might make the statistics in the evaluation inaccurate. To reduce this threat, we carefully selected detection tools. The employed tools (e.g., *PDM*, *Metrics*, *JDeodorant*) are widely used and acknowledged. *UselessDect* developed by the authors had been intensively tested before it was employed in the evaluation for the detection of *Useless Field*, *Useless Method*, and *Useless Class*.

6 Discussion

6.1 Target Scenario and Practicability

As discussed in Sections 1 and 4, the recommended resolution sequence of bad smells can be used if and only if refactorings are performed in batch model. To investigate the practicability of the resolution sequence, we would investigate how often refactorings are conducted in batch model.

Software refactoring can be carried out in two scenarios. The first is XP-style small-step refactoring, which is called floss refactoring [19]. To implement a new function, XP encourages engineers to quickly complete an initial implementation that can pass predefined test cases. And then XP encourages engineers to improve the implementation through refactoring. The refactoring is usually confined to a small scope (a few local files), and bad smell detection tools are rarely used. This is not a target scenario of this paper. However, the pairwise analysis in Section 3 remains useful because overlapping bad smells may also be encountered in this scenario.

The other refactoring scenario is that in which an application is thoroughly refactored in one attempt. This is what we call batch model. Murphy-Hill and Black [19] called it root canal refactoring. It is usually employed in software maintenance [24] (as depicted in Section 1). It is also intensively employed between two consecutive development iterations in iterative development. For example, Microsoft typically reserves 20 percent of development efforts on thorough refactoring right after an application is released and before the development of the next version is initiated [25]. Generally, software development teams creating a single product iteratively expend more time on thorough refactoring. Newegg,[10] a well-known international e-business company, is a good example. Software development teams of the company spend most of their time developing and maintaining the e-business system. Once a new version of the system is released, development of the next version does not start immediately. Instead, time is allocated for the evaluation of the new release and analysis of new requirements. During the period, some software engineers fix bugs, but more are assigned to thoroughly refactoring the application. Refactoring improves the extensibility of the application, and once new features (requirements) of the next version are determined these features can be easily and quickly added to the existing version. Fast delivery is critical for highly competitive industries, such as e-business. Apart from the cases of Microsoft and Newegg mentioned above, similar cases of batch model refactoring have also been reported [26], [27]. Batch model refactoring is also the target scenario of most existing bad smell detection tools [1], [10], [28], [29], [30], [22], [31], [32], [33]. As discussed in the preceding paragraph, bad smell detection tools are rarely used in small-step floss refactoring. Therefore, the popularity of these detection tools suggests that refactoring in batch model is popular.

The recommended resolution sequence is only for reference. Different kinds of bad smells do not have to be constantly resolved strictly using the recommended resolution sequence. For the following special cases, the resolution sequences should be rescheduled.

- First, there is no sufficient time to resolve all bad smells. In this case, the most severe bad smells should be resolved first [4]. But if possible, we recommend resolving all bad smells in the recommended sequence to achieve greatest quality improvement with least refactoring effort.
- Second, software engineers plan to improve certain quality attributes by refactoring. They may realize that their software applications are poor in terms of certain quality attributes, e.g., *cohesion*. Consequently, they plan to improve *cohesion* by refactoring. In this case, bad smells having the greatest negative impact on *cohesion* should be detected and resolved first.

6.2 Why Informal Analysis

The main weakness of the paper is that bad smell detection and resolution sequences are produced by informal analysis, while formal analysis and reasoning are commonly expected in research papers. The formal analysis on bad smell detection and resolution sequences is currently unavailable for the following reasons:

- First, bad smells have not yet been completely formalized. Currently, bad smells are defined using informal or semiformal descriptions [3], [4], [23]. For example, *Large Class* sounds simple and it is widely accepted that the larger a class is, the harder it is to maintain. However, the criterion of *Large Class* is not

10. http://www.newegg.com.

as easy to define as it seems: How large is *Large*? If a single class contains 10,000 lines of source code, we are sure that it is a *Large Class*. However, what of 200, 100, or 50 lines? Should other metrics such as *cyclomatic complexity* and *cohesion* be taken into consideration? If so, what are their thresholds?

- Second, detailed solutions to bad smells have not yet been formalized. As suggested in Fig. 1, solutions to bad smells have to be manually created by software engineers. Although bad smells are usually associated with certain kinds of refactoring rules, manual decisions need to be made before the refactoring rules can be applied. For example, refactoring rule *Extract Method* can be applied to methods to resolve *Long Method*. However, the statements of the long method that require extraction have to be decided before *Extract Method* can be applied. But no widely accepted formal algorithm is available to make the decision automatically.

The informality of bad smells, together with the uncertainty in their solutions, makes it challenging to formally analyze the impact of bad smell resolution, let alone their interactions and resolution sequences.

Although researchers have succeeded in formalizing refactorings [34], [35], [36], [8], [37], [38], [39], e.g., *Pull up Method*, the formalization of bad smells and their solutions is an entirely different matter.

6.3 Symptom versus Disease

As suggested by Trifu and Marinescu [40], bad smells are symptoms of design problems. Different design problems may lead to the same symptoms (bad smells), but their solutions may differ. Consequently, the solution to a specific kind of bad smell may vary according to the design problems underneath. As a result, the detection and resolution sequences of bad smells may be conditional: In some situations a certain resolution sequence is preferred, whereas in other situations the reverse may be better.

A good solution to this problem is to scheme the detection and resolution of design problems instead of their symptoms (bad smells). However, *design problems* have not yet attracted as much attention and tool support as have *bad smells*. Once *design problems* are widely accepted and well supported by tools, devising a scheme for their detection and resolution would be an interesting research direction.

As a compromise, we attempt to subdivide bad smells that may need completely different solutions depending on their contexts. A good example is *Primitive Obsession*. In some cases, its resolution may change the fields of its enclosing class. In these cases, *Primitive Obsession* should be resolved before *Long Parameter List* (see Section 3.3.14 for details). However, in other situations, the resolution of *Primitive Obsession* would not change the fields of its enclosing class. In these cases, *Primitive Obsession* and *Long Parameter List* can be detected and resolved in any order. Consequently, we subdivide *Primitive Obsession* into *Simple Primitive Obsession*, *Simple Type Code*, and *Complex Type Code*. The resolution of the last subtype (*Complex Type Code*) would change the fields of its enclosing class, whereas the other two would not. The division makes it easier to optimize resolution sequences.

6.4 Pairwise Analysis

The analysis of bad smell detection and resolution sequences is pairwise, and does not examine the cases in which more than two bad smells appear in the same source code entity. There are too many $(C_{11}^3 = 990)$ triple combinations of the analyzed 11 kinds of bad smells, even more $(C_{11}^4 = 7{,}920)$ quadruple combinations. Because the formal analysis of bad smells is not yet available (see Section 6.2), it is impractical to manually analyze all the cases.

6.5 Granularity and Optimization

As discussed in Section 4, the scheme of bad smell detection and resolution is confined to the *kind-level*, i.e., all bad smells of the same kind are scheduled as a single unit. The schedule is not optimized according to applications that require refactoring. The schedule would be more effective if it can be conducted in smaller grain and optimized according to applications to be refactored.

The first technique (lightweight) of creating a small-grained application-optimized scheme is the two-level scheme (Fig. 1). Kind-level should be scheduled first, and then instances of a specific kind of bad smell are scheduled. The instance-level scheme is application-optimized, based on the distribution of instances of bad smells. The kind-level scheme is the target of this paper, but instance-level is out of the scope. One of our potential future directions is to scheme instances of a specific kind of bad smell, e.g., resolution sequence of all clones found in a specific application [21].

An alternative (heavyweight) to the fine-grained application-optimized scheme is a flat scheme on all bad smells detected in a specific application. All kinds of bad smells are detected first, and then their resolution sequences are managed. We applied this in our previous work [41], [42], but found it incompatible with existing tools. This approach requires tools to detect bad smells, mark them, and apply refactorings sometime later (after all available refactorings have been detected). But existing tools do not support this approach. Usually a detection tool detects a specific kind of bad smell,and software engineers check and refactor it before the next detection tool is picked up. Consequently, to make flat scheme practical, existing tools have to be revised to store reported bad smells in a central repository and restore them after their resolution sequences are determined. It is a considerable challenge to conduct such a revision on various existing tools (detection and refactoring tools).

Detecting all kinds of bad smells with a single framework (tool) has also been investigated [30], [40]. In this case, bad smells are usually associated with detection criteria (described in formal languages) [30], [40] which are similar to formalized preconditions of refactorings [8]. And then, their detection algorithms are automatically generated [30], [40]. With such a framework, the kind-level scheme proposed in this paper remains practical. However, a flat application-optimized scheme would generate better results. In our previous work [41], [42], we went a step further by associating bad smells with formalized refactoring rules and finally scheduled the application of conflicting refactorings (instead of overlapping bad smells). One of the potential problems with these all-purpose detection

frameworks is the formalization of detection criteria for complex bad smells and generation of detection algorithms for these bad smells. For example, to the best of our knowledge, no existing framework can formalize clone detection criteria and automatically generate practical clone detection algorithms comparable to those manually designed by experts.

6.6 Extensibility

As an initial study, we only considered some (not all) kinds of existing bad smells. An increasing number of bad smells are constantly being discovered. Additional bad smells can be analyzed in the same manner as evaluated bad smells were analyzed in this paper. If a new kind of bad smell is added, software engineers should analyze its interaction with analyzed bad smells, and recommend its detection and resolution sequences relative to the analyzed bad smells. With these relative sequences, the final resolution sequence could be produced by rerunning the algorithm in Section 4.

A generic suggestion on analyzing the resolution sequence of a pair of bad smells is presented in Section 3.1. However, analysis following the suggestion is inevitably subjective because the suggestion is not a formal approach. Furthermore, five questions should be answered during the analysis, and answers to different questions may yield conflicting resolution sequences. Analysts should compare the frequencies and severities of different interactions between bad smells (each question asked in Section 3.1 investigates an interaction). Finally, additional interactions aside from the five listed in Section 3.1 might be found.

7 Related Work

7.1 Bad Smells

To specify what kind of source code should be restructured, Fowler et al. [3] proposed the concept of *bad smells*. They proposed and described 22 bad smells in object-oriented systems. They also associated refactoring rules with these bad smells, suggesting how to resolve these bad smells.

Bad smells in specific domains have also been proposed. Srivisut and Muenchaisri [43] defined some bad smells in aspect-oriented software, and proposed approaches to detect them. Van Deursen et al. [44] described 11 *Test Smells* indicating problems in test code.

The impact of bad smells has also been analyzed. Lozano et al. [45] assessed the impact of bad smells, i.e., the extent to which different bad smells influence software maintainability. They argued that it is possible to analyze the impact of bad smells by analyzing historical information. With the impact in mind, it is possible to assess code quality by detecting and visualizing bad smells. Van Emden and Moonen [46] implemented a code browser for detecting and visualizing code smells, and assessed the quality of code according to the visual representation.

Detecting bad smells is critical and time-consuming. Therefore, automating detection is essential. Tsantalis et al. [47] proposed an approach to identifying and removing type-checking bad smells which is implemented in an prototype tool named *JDeodorant*. Fokaefs et al. [48] proposed an Eclipse plug-in to identify and resolve feature envy bad smells. Clones, one of the most common bad smells, have been investigated for a long time, and dozens of detection algorithms have been proposed [9] to detect them. Moha et al. [29] proposed a language for formalizing bad smells, and a framework for automatically generating detection algorithms for the formalized bad smells. Tourwe and Mens [31] formalized bad smells with logic metaprogramming and detected bad smells automatically. Munro [33] related bad smells to software metrics to detect bad smells with automatically collected metrics. Van Rompaey et al. [32] proposed a set of metrics to detect test smells for unit testing. However, Mantyla et al. [49] argued that detecting bad smells using metrics is difficult. In their experience, manual evaluation of bad smells does not correlate to source code metrics. Other detection algorithms can be found in the survey by Mens and Tourwe [1].

Trifu et al. [23] proposed a quality-guided approach to detect and remove design flaws. In their approach, a quality model is first proposed, and only those design flaws negatively impacting the quality model are detected. For each kind of design flaw, all alternative solutions are predesigned and their impacts on quality attributes are predefined. Once a design flaw is detected and confirmed, its alternative solutions are checked and the one that has greatest positive impact on software quality (according to the predefined quality model) is selected. The paper [23] differs from ours in that it focuses on which kind of bad smells should be detected and which refactorings should be applied, whereas this paper focuses on the sequence in which different kinds of bad smell should be detected and resolved.

Trifu and Marinescu [40] distinguished structural symptoms (code smells) from structural problems (design problems). They argued that properly defined high-level design problems could be diagnosed by detecting related bad smells, and these design problems can be mapped to unique refactoring solutions.

7.2 Relationships among Bad Smells

Relationships among bad smells have also ben investigated. Wake [4] classified bad smells into two categories: bad smells within classes and bad smells between classes. Meszaros [50] classified test smells into *code smells*, *behavior smells*, and *project smells*.

Mantyla et al. [51] analyzed the correlations among bad smells by investing the frequency with which each pair of bad smells appears in the same module. They found that bad smells within the same category are more likely to appear together. The work aimed to simplify the comprehension of bad smells, instead of refactoring activities. The authors did not analyze the interinfluence of the resolutions of different bad smells, or the resolution sequences of bad smells.

Pietrzak and Walter [52] investigated the intersmell relationships to facilitate the detection of bad smells. They argued that detected or rejected bad smells might imply the existence or absence of other bad smells. Their work aimed to simplify the detection of bad smells, whereas our work focuses on the detection and resolution sequences of different kinds of bad smells.

7.3 Scheme of Refactorings

In previous work, we attempted to schedule refactorings [41], [42]. We first listed all available refactorings for a given

system, and then arranged an application sequence for these refactorings to maximize the effect of refactoring activity. A problem with this approach is that it does not collaborate well with existing refactoring tools as discussed in Section 6.

To make the scheme compatible with existing refactoring methods, this paper tries to schedule the resolution of different kinds of bad smells. In this way, different kinds of bad smells are resolved one after the other. It cooperates well with existing refactoring tools: A detection tool is chosen to detect a specific kind of bad smell, the bad smells are resolved, and then the next detection tool is selected.

Wake [4] suggested resolving the most severe bad smells first, but he did not define the severity of bad smells. Furthermore, the strategy assumes that all reported bad smells will not be resolved. Software refactoring has been proved worthwhile, and we suggest carefully resolving all bad smells.

Bouktif et al. [21] attempted to schedule clone refactoring activities. Because of resource limitation, not all clones can be removed. Consequently, they tried to select a subset of detected clones resolving which would lead to the greatest quality improvement while resource consumption is minimized. As illustrated in Fig. 1, the scheme proposed in this paper and the one proposed by Bouktif et al. [21] are on different levels: The latter focuses on the resolution sequence of different occurrences of the same kind of bad smells (clone), whereas we focus on the detection and resolution sequence of different kinds of bad smells.

Mens et al. [53] realized that once a badly structured code is detected, many refactorings would be proposed. In this case, software engineers should decide which refactorings should be applied and in which sequence. To facilitate the decision, Mens et al. [53] proposed a critical-pair-based approach to analyze refactoring dependencies. Their work [53] differs from ours in that they focused on XP style small step refactoring, i.e., a small code unit is checked manually by its author and all bad smells within the unit would be detected all at once. In this case, all bad smells can be resolved as a whole, whereas we aim at the situation in which a large system is thoroughly refactored in one attempt. In this case, tool support is indispensable in bad smell detection. Thus, different kinds of bad smells have to be detected and resolved one after the other.

In previous work, we have briefly discussed the resolution sequences of bad smells [54], but no evaluation or discussion was presented. This paper is an expanded version of that [54]. In this paper, an evaluation on two nontrivial applications is presented. Furthermore, the need for resolution sequences is illustrated in detail with a motivating example.

Research in other aspects of software refactoring can be found in the survey by Mens and Touwe [1].

8 Conclusion

In this paper, we first motivated the necessity of arranging resolution sequences of bad smells with an example. Then, we illustrated how to arrange such a resolution sequence for commonly occurring bad smells. We also carried out evaluations on two nontrivial applications to validate the research. The results suggest a significant reduction in refactoring effort ranging from 17.64 to 20 percent can be achieved when bad smells are resolved using the recommended resolution sequence.

The contributions of this paper are as follows: First, it discovers and illustrates the importance of resolution sequences of bad smells. Second, it proposes a resolution sequence for commonly occurring bad smells. Finally, it validates the effect of resolution sequences of bad smells on two nontrivial applications.

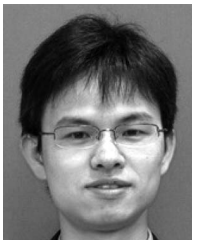
Acknowledgments

The authors would like to say thanks to the students in the laboratory. They did most of the work in the evaluation. The authors would like to say thanks to Dr. Lisa Liu from ETH Zurich for her help in preparing this manuscript. The authors also thank the anonymous reviewers for their valuable suggestions. Constructive comments and suggestions from ESEC/FSE'09 reviewers on how to revise and extend the original version are highly appreciated as well. The work is funded by the National Natural Science Foundation of China No. 61003065 and 60773152, the National High-Tech Research and Development Plan of China No. 2007AA010301, Specialized Research Fund for the Doctoral Program of Higher Education (No. 20101101120027), and Excellent Young Scholars Research Fund of Beijing Institute of Technology (No. 2010Y0711).

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] W.C. Wake, *Refactoring Workbook.* Addison Wesley, Aug. 2003.
[5] http://wiki.java.net/bin/view/People/SmellsToRefactorings, 2011.
[6] 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.
[7] F. Tip, A. Kiezun, and D. Baeumer, "Refactoring for Generalization Using Type Constraints," *Proc. 18th Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 13-26, Oct. 2003.
[8] T. Mens, N.V. Eetvelde, and S. Demeyer, "Formalizing Refactorings with Graph Transformations," *J. Software Maintenance and Evolution: Research and Practice,* vol. 17, no. 4, pp. 247-276, 2005.
[9] R. Koschke, "Identifying and Removing Software Clones," *Software Evolution,* T. Mens and S. Demeyer, eds., pp. 15-36, Springer, 2008.
[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654-670, July 2002.
[11] 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.
[12] 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.
[13] 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.
[14] R. Wettel and R. Marinescu, "Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments," *Proc. Seventh Int'l Symp. Symbolic and Numeric Algorithms for Scientific Computing,* p. 63, 2005.

[15] Eclipse Foundation. Eclipse 3.4.2. http://www.eclipse.org/emft/projects/, 2011.
[16] Microsoft Corporation. Microsoft Visual Studio 2008. http://www.microsoft.com/, 2011.
[17] JetBrains Company. IntelliJ IDEA 8. http://www.jetbrains.com/idea/, 2011.
[18] E. Mealy and P. Strooper, "Evaluating Software Refactoring Tool Support," *Proc. Australian Software Eng. Conf.*, pp. 331-340, 2006.
[19] E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software,* vol. 25, no. 5, pp. 38-44, Sept./Oct. 2008.
[20] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving Usability of Software Refactoring Tools," *Proc. Australian Software Eng. Conf.*, pp. 307-318, Apr. 2007.
[21] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A Novel Approach to Optimize Clone Refactoring Activity," *Proc. Eighth Ann. Conf. Genetic and Evolutionary Computation,* pp. 1885-1892, July 2006.
[22] 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.
[23] A. Trifu, O. Seng, and T. Genssler, "Automated Design Flaw Correction in Object-Oriented Systems," *Proc. Eighth Euromicro Working Conf. Software Maintenance and Reeng.*, pp. 174-183, 2004.
[24] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and Y. Kataoka, "On Refactoring for Open Source Java Program," *Proc. Australian Conf. Software Measurement*, pp. 247-251, 2003.
[25] M.A. Cusumano and R.W. Selby, *Microsoft Secrets.* Free Press, 1995.
[26] E. Stroulia and R. Kapoor, "Metrics of Refactoring-Based Development: An Experience Report," *Proc. Seventh Int'l Conf. Object-Oriented Information System,* pp. 113-122, 2001.
[27] R. Weber, T. Helfenberger, and R.K. Keller, "Fit for Change: Steps Towards Effective Software Maintenance," *Proc. IEEE 21st Int'l Conf. Software Maintenance—Industrial and Tool Vol.,* pp. 26-33, 2005.
[28] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated Support for Program Refactoring Using Invariants," *Proc. IEEE 17th Int'l Conf. Software Maintenance,* pp. 736-743, 2001.
[29] N. Moha, Y.-G. Gueheneuc, and P. Leduc, "Automatic Generation of Detection Algorithms for Design Defects," *Proc. IEEE/ACM 21st Int'l Conf. Automated Software Eng.*, pp. 297-300, Sept. 2006.
[30] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "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.
[31] T. Tourwe and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," *Proc. Seventh European Conf. Software Maintenance and Reeng.*, pp. 91-100, 2003.
[32] 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.
[33] M. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," *Proc. IEEE 11th Int'l Symp. Software Metrics,* p. 15, Sept. 2005.
[34] M.L. Cornelio, "Refactorings as Formal Refinements," PhD thesis, Federal Univ. of Pernambuco, 2004.
[35] R. Gheyi and T. Massoni, "Formal Refactorings for Object Models," *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 208-209, 2005.
[36] T. Mens, S. Demeyer, and D. Janssens, "Formalising Behaviour Preserving Program Transformations," *Proc. Int'l Conf. Graph Transformation,* pp. 286-301, 2002.
[37] T. Mens, G. Taentzer, and O. Runge, "Detecting Structural Refactoring Conflicts Using Critical Pair Analysis," *Electronic Notes in Theoretical Computer Science,* vol. 127, no. 3, pp. 113-128, Apr. 2005.
[38] V.E.N and Janssens, "Extending Graph Transformation for Refactoring," *Proc. Second Int'l Conf. Graph Transformation,* pp. 399-415, Sept./Oct. 2004.
[39] I. Porres, "Model Refactorings as Rule-Based Update Transformations," *Proc. Sixth Int'l Conf. Unified Modeling Language,* pp. 59-174, 2003.
[40] A. Trifu and R. Marinescu, "Diagnosing Design Problems in Object Oriented Systems," *Proc. 12th Working Conf. Reverse Eng.*, pp. 155-164, 2005.
[41] H. Liu, G. Li, Z. Ma, and W. Shao, "Scheduling of Conflicting Refactorings to Promote Quality Improvement," *Proc. IEEE/ACM 22nd Int'l Conf. Automated Software Eng.*, pp. 489-492, 2007.
[42] H. Liu, G. Li, Z. Ma, and W. Shao, "Conflict Aware Scheduling of Software Refactorings," *IET Software,* vol. 2, no. 5, pp. 446-460, Oct. 2008.
[43] K. Srivisut and P. Muenchaisri, "Defining and Detecting Bad Smells of Aspect-Oriented Software," *Proc. 31st Ann. Int'l Computer Software and Applications Conf.,* vol. 1, pp. 65-70, July 2007.
[44] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring Test Code," *Proc. Second Int'l Conf. Extreme Programming and Flexible Processes in Software Eng.*, May 2001.
[45] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the Impact of Bad Smells Using Historical Information," *Proc. Ninth Int'l Workshop Principles of Software Evolution: In Conjunction with the Sixth ESEC/FSE Joint Meeting,* pp. 31-34, 2007.
[46] E. van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," *Proc. Ninth Working Conf. Reverse Eng.*, pp. 97-106, 2002.
[47] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and Removal of Type-Checking Bad Smells," *Proc. 12th European Conf. Software Maintenance and Reeng.*, pp. 329-331, Apr. 2008.
[48] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and Removal of Feature Envy Bad Smells," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 519-520, Oct. 2007.
[49] M. Mantyla, J. Vanhanen, and C. Lassenius, "Bad Smells—Humans as Code Critics," *Proc. IEEE 20th Int'l Conf. Software Maintenance,* pp. 399-408, Sept. 2004.
[50] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, 2007.
[51] M. Mantyla, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," *Proc. Int'l Conf. Software Maintenance,* pp. 381-384, 2003.
[52] B. Pietrzak and B. Walter, "Leveraging Code Smell Detection with Inter-Smell Relations," *Proc. Seventh Int'l Conf. Extreme Programming and Agile Processes in Software Eng.*, pp. 75-84, June 2007.
[53] T. Mens, G. Taentzer, and O. Runge, "Analysing Refactoring Dependencies Using Graph Transformation," *Software and Systems Modeling,* vol. 6, no. 3, pp. 269-285, Sept. 2009.
[54] H. Liu, L. Yang, Z. Niu, Z. Ma, and W. Shao, "Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells," *Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. the Foundations of Software Eng.*, pp. 265-268, 2009.



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.

Zhiyi Ma is an associate professor in the Institute of Software, School of Electronics Engineering and Computer Science at Peking University. His current research interests include service-oriented computing, model-driven development, and metamodeling technology. As a project leader or main member, he has engaged in more than 20 research projects.

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.

Zhendong Niu received the PhD degree in computer science from the Beijing Institute of Technology in 1995. He was a postdoctoral researcher at the University of Pittsburgh from 1996 to 1998, and a researcher/adjunct faculty member at Carnegie Mellon University from 1999 to 2004, and a joint research professor in the Information School at University of Pittsburgh from 2006. He received the IBM Faculty Innovation Award in 2005 and was awarded the New Century Excellent Talents in University of MOE of China in 2006. He is a professor and deputy dean of the School of Computer Science and Technology at the Beijing Institute of Technology. His research areas include digital libraries, web-based learning techniques, informational retrieval, software architecture, etc. He serves as an editorial board member for international journal of learning technology, etc. He has published more than 80 papers in journals and international conferences in his fields.