Autom Softw Eng (2013) 20:81–110
DOI 10.1007/s10515-012-0100-0

Identification of generalization refactoring opportunities

Hui Liu · Zhendong Niu · Zhiyi Ma · Weizhong Shao

Received: 25 July 2011 / Accepted: 8 February 2012 / Published online: 1 March 2012

Abstract Generalization refactoring helps relate classes and share functions, including both interfaces and implementation, by inheritance. To apply generalization refactoring, developers should first identify potential generalization refactoring opportunities, i.e., software entities that might benefit from generalization refactoring. For non-trivial software systems, manual identification of these opportunities is challenging and time-consuming. However, to the best of our knowledge, no existing tools have been specifically designed for this task. As a result, people have to identify these opportunities manually or with the help of tools designed for other purposes, e.g., clone detectors. To this end, we propose a tool *GenReferee* (*Generalization Referee*) to identify potential refactoring opportunities according to conceptual relationship, implementation similarity, structural correspondence, and inheritance hierarchies. It was first calibrated on two non-trivial open source applications, and then evaluated on another three. Evaluation results suggest that the proposed approach is effective and efficient.

Keywords Software quality · Software refactoring · Generalization refactoring · Opportunity identification · Tool support · Software maintenance

H. Liu (✉) · Z. Niu
School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
e-mail: liuhui08@bit.edu.cn

Z. Niu
e-mail: zniu@bit.edu.cn

Z. Ma · W. Shao
Key Laboratory of High Confidence Software Technologies, Peking University, Ministry of Education, 100871, Beijing, China

Z. Ma
e-mail: mzy@sei.pku.edu.cn

W. Shao
e-mail: wzshao@pku.edu.cn

1 Introduction

Generalization is one of the widely used dominant features of object-oriented design and programming. It helps to share common functions, including both interfaces and implementation. Sharing common interfaces and their implementation by generalization has the following benefits. First, it would provide users a unified convenient access to these functions. Second, it helps to avoid inconsistency among these interfaces during software evolution. Finally, it would reduce duplicate code that has been recognized as one of the severest bad smells (Fowler et al. 1999). Although sharing common interfaces only would not remove duplicate code (the third benefit), it might bring the first two benefits as well.

Generalization Refactoring, also known as *Refactoring for Generalization* (Fowler et al. 1999; Tip et al. 2003) and *Refactoring to Generalize* (Opdyke 1992), is to restructure object-oriented source code to make full use of generalization. The importance of generalization refactoring has been investigated (Tip et al. 2003; Adnan et al. 2008; Cottrell et al. 2007; Streckenbach and Snelting 2004; Steimann 2007; Opdyke 1992; Moore 1995).

One of the key steps in generalization refactoring is to identify potential refactoring opportunities. For non-trivial systems, fully manual identification is challenging and time-consuming. However, to the best of our knowledge, no existing approaches or tools have been specially designed for this purpose, and thus these approaches (and their corresponding tools) fail to solve this issue. Some approaches have been proposed to optimize class hierarchies according to given client programs (Streckenbach and Snelting 2004; Steimann 2007). But their main purpose is to minimize the coupling between source code and its client programs by introducing smaller interfaces. As a result, they fail to improve internal structural of the source code, and internal generalization refactoring opportunities are not identified. Moore (1995) proposed a tool *Guru* to detect duplicate methods (with identical signatures and identical implementation), and remove these methods by generalization refactoring. But similar methods, slightly modified in signature or implementation, are missed.

Lacking effective tool support, developers have to identify generalization refactoring opportunities manually or with the help of tools originally designed for other purposes (Cottrell et al. 2007), e.g., clone detection tools. Duplicate fragments distributed in sibling classes might benefit from generalization refactoring (Higo et al. 2005; Higo et al. 2004). Consequently, developers might find some generalization refactoring opportunities from clone detection results. This is one of the reasons why some developers employ clone detectors to identify generalization refactoring opportunities. However, to remove clones, *delegation* is usually a better choice than *generalization* (Kegel and Steimann 2008). As a result, most clone sets reported by clone detection tools are not good generalization refactoring opportunities. In our evaluation reported in Sect. 7, more than 95% of the reported clone sets were not taken by developers as generalization refactoring opportunities. *Aries* proposed by Higo et al. (2004) tries to filter and classify results (clone sets) of *CCFinder* according to refactorings that can be applied to remove clone sets. With the classification, *Aries* might improve its accuracy in identifying generalization refactoring opportunities. However, as suggested by evaluation results in Sect. 7, more than 70% of generalization refactoring opportunities are filtered by *Aries*. One of the possible reasons is that the filter is originally designed for clone resolution, instead of generalization opportunity identification. Furthermore, clone-based tools, including both *CCFinder* and *Aries*, might miss generalization opportunities that do not contain clones. According to our initial evaluation in Sect. 7, more than 30% of generalization opportunities do not contain clone sets. It is possible that these opportunities could not be identified by clone-based identification tools.

To this end, we propose an approach in this paper to identify generalization refactoring opportunities. The approach analyzes conceptual relationship, implementation similarity, and structural correspondence among classes, as well as their inheritance hierarchies. According to the analysis, the approach recommends a list of refactoring opportunities. The approach is semi-automatic and interactive in that the recommended opportunities should be checked manually by software engineers, and the final decision is up to them. Developers might take the suggested opportunities as they are, or they might make modifications to fine-tune the suggested opportunities. Compared to manual identification and clone-based detection, the proposed approach can dramatically reduce the amount of classes to be checked manually by software engineers. Consequently, a great reduction in effort could be expected.

The proposed approach has been implemented and evaluated. It is implemented as an *Eclipse* plug-in, *GenReferee* (*Generalization Referee*). With the implementation, we have conducted an experiment to evaluate the proposed approach. The evaluation consists of two parts. In the first part, the approach is calibrated on two non-trivial open source applications. In the second part, the approach is evaluated on another three applications, and compared to fully manual detection, and clone-based detection. Evaluation results suggest that the proposed approach is effective and efficient.

The rest of this paper is structured as follows. Section 2 discusses related works. Section 3 describes a sample scenario illustrating why and how *GenReferee* is used to recommend refactoring opportunities. Section 4 presents an overview of the approach and details of the approach are presented in Sects. 5–6. Section 7 reports the experiment, and Sect. 8 discusses some issues. Finally Sect. 9 concludes the paper.

2 Related work

2.1 Refactorings for generalization

Opdyke (1992) explained generalization refactorings, and formalized their preconditions. Fowler et al. (1999) made a further explanation of these refactorings.

Tip et al. (2003) proposed an approach to verify preconditions of generalization refactorings and determine allowable modifications. Type constraints generated automatically by their approach have been proved to be useful, and the approach has been implemented in *Eclipse*. Once *GenReferee* identifies refactoring opportunities, type constrains are helpful in determining allowable modifications. Consequently, *GenReferee* and type constraints may work together, fulfilling different sub-tasks of generalization refactoring.

Moore (1995) proposed a tool *Guru* to restructure inheritance hierarchies. The purpose of *Guru* is to remove duplicate methods by adjusting inheritance hierarchies, e.g., adding new superclasses. Consequently, it only deals with duplicate (identical) methods (with identical signatures and identical implementation), whereas *GenReferee* proposed in this paper deals with similar methods and duplicate attributes, as well as duplicate methods.

Streckenbach and Snelting (2004) proposed *KABA* to optimize class hierarchies according to a given set of client programs. The main purpose of their work is to minimize the interfaces of classes referred in the client programs by extracting new smaller classes (as new interfaces) and retyping the references. This differs from *Genreferee* that aims to share common functions among classes. In other words, though some internal classes that are not directly invoked by the given client programs might benefit from generalization refactoring, they could not be identified by *KABA*. Because *KABA* optimized inheritance hierarchies according to client programs, it is difficult to apply *KABA* to frameworks, e.g., Java SKD, whose clients are not specified.

Snelting and Tip (1998) reengineer class hierarchies using concept analysis. By analyzing the usage of the class hierarchy by some specific applications, they built a concept lattice for the variables and class members. In such as way, they can find some design bad smells, e.g., *useless member* and *divergent change*. Their work focuses more on pulling down class members, and splitting large classes. In contrast, *Genreferee* aims to sharing members among classes via generalization refactorings.

Steimann (2007) proposed an approach to determine how to extract interfaces and how to use them. The purpose of his work is to decouple clients from their concrete servers by introducing interfaces. For a given variable or object reference, it computes the most general type (interface) that can be used for this reference without type casts. His work focused on inferring interfaces for individual classes, whereas *GenReferee* focuses on finding commonalities (in implementation or signature) among different classes.

Higo et al. (2005, 2004) proposed a refactoring assistant *Aries* to characterize code clone with metrics. One of these metrics, *Dispersion of Class Hierarchy* (*DCH*), could assist developers to decide how to remove code clone, i.e., by *Extract Method* or *Pull Up Method*. If all fragments of a clone set appear within the same class ($DCH = 0$), *Extract Method* would be recommended. Otherwise, *Extract Method* would be recommended. Other metrics, e.g., *NRV*(*Number of Referred Variables*) and *NAV* (*Number of Assigned Variables*) are presented to assist developers to select appropriate structures to be extract. This is similar to the work published by (Yang et al. 2009; Tsantalis and Chatzigeorgiou 2011). Essentially, *Aries* is a clone-detection based refactoring assistant (depends on *CCSharper* (Higo et al. 2002) and *CCFinder* (Kamiya et al. 2002)). *GenReferee* and *Aries* overlap in that both of them might be used to recommend generalization refactoring opportunities. But *Aries* focuses on how to resolve a clone set by *Pull Up Method* or *Extract Method* (it is not a generalization refactoring), whereas *GenReferee* focuses on how to identify pairs of classes that would benefit from generalization refactoring. Evaluation results in Sect. 7 suggest that it might not be a good idea to identify generalization refactoring opportunities with *Aries*.

2.2 Resolution of structural correspondence

Cottrell et al. (2007) proposed *Breakaway* to resolve structural correspondence between a pair of intensively related classes. It is proved to be helpful in determining how to refactor a pair of classes via generalization refactoring. Our work focuses on the preceding step of what it focuses: how to detect classes that may deserve generalization refactoring, i.e., how to select classes to be checked by *Breakaway*. Consequently, *Breakaway* and our tool *GenReferee* can cooperate to facilitate generalization refactorings.

While identifying refactoring opportunities, *GenReferee* also resolves structural correspondence between candidate classes. This is similar to that of *Breakaway* (Cottrell et al. 2007). They differ in that:

– Purpose and emphasis. Resolving structural correspondence is all that *Breakaway* does for a pair of selected classes. But the resolution is not the ultimate objective of *GenReferee*. This is a preparation for the computation of the strength of recommendation. The ultimate objective of *GenReferee* is to recommend refactoring opportunities. Furthermore, classes selected by *GenReferee* would be rechecked by *Breakaway* later. Therefore, *Breakaway* focuses on accuracy, whereas the resolution of structural correspondence in *GenReferee* focuses more on efficiency.
– Implementation comparison. *GenReferee* compares implementation by *diff*, whereas *Breakaway* employs a greedy algorithm.
– Overriding methods. *GenReferee* resolves correspondence of overriding methods with signatures and inheritance information. This is more accurate and efficient than *Breakaway* relying on implementation comparison. In our experiment, some errors have been made by *Breakaway* in correlating overriding methods. For example, structural correspondence among some override methods of classes *ResizableHandleKit* and *NonResizableHandleKit* in *GEF* have not been resolved correctly.
– Method signatures. Method signatures are compared by *Breakaway* as common AST nodes. This may lead to some mismatches of overloading methods. *GenReferee* distinguishes signature nodes from implementation nodes, and applies a semantic sensitive algorithm on signature comparison.

A special kind of structural correspondence resolution is to match program elements between two versions of a program. This kind of matching is critical for evolution analysis, e.g., API evolution, version merging, and regression testing. It differs *GenReferee* in that it resolves structural correspondence between successive versions of the same application, whereas *GenReferee* and *Breakaway* focus on a single version. As a result, some useful information in comparing entities between successive versions, e.g., relationships among entities (Xing and Stroulia 2005) and change rules (Kim et al. 2007; Kim and Notkin 2009), would be less helpful, if not useless, in resolving correspondence among elements of the same version. But some technologies used to match successive versions might be useful for resolving structural correspondence among different classes. As suggested by Kim and Notkin (2006), techniques for matching program elements between successive versions could be classified into eight categories according to adopted program representation. The fist category is entity name matching, which matches entities simply by entity names (Zimmermann et al. 2005). The second category is string matching, which consumes that entities containing the *Longest Common Subsequence* (*LCS*) match. *GenReferee* employs *Diff* (Hunt 1976) in calculating implementation similarity, and *Diff* is a well-known implementation of a string match algorithm. Besides implementation similarity, *GenReferee* also calculates other similarity, e.g., signature similarity, to assist resolution of structural correspondence. Others technologies discussed by Kim and Notkin (2006) include syntax tree matching (Yang 1991; Hunt and Tichy 2002), control flow graph matching (Laski and Szermer 1992), binary code matching (Wang et al. 2000), ant those for clone detection (Baker 1992) and origin analysis (Demeyer et al. 2000; Godfrey and Zou 2005; Dig et al. 2006).

Revelle et al. (2011) proposed a *Latent Semantic Indexing* (*LSI*) based approach to calculate the similarity between software entities, e.g., methods and classes. It extracts identifiers and comments from source code as a vocabulary set. With this vocabulary set, text similarity between entities containing these vocabularies could be calculated by *LSI*. *GenReferee*' rough selection based on conceptual relationship is a simplified version of this approach.

While resoling structural correspondence, existing approaches usually depend on identifier splitting and expanding. Feild et al. (2006), Enslen et al. (2009), Guerrouj et al. (2011), Madani et al. (2010) have proposed approaches to split identifiers in source code, and these approaches might be useful for *GenReferee* while camel-case or Pascal-case styles are not strictly followed.

2.3 Clone detection

Currently, clone detection tools, e.g., *CCFinder*, are used to identify generalization refactoring opportunities (Cottrell et al. 2007). Experimental comparison between *CCFinder* and *GenReferee* has been presented and discussed in Sect. 7. Experimental results suggest that the proposed approach is more accurate and efficient than clone-based approach.

Rough selection in Sect. 5.1 is similar to clone detection algorithms. The difference is explained as follows. First, rough selection in our approach selects classes according to both conceptual relations and implementation similarity, whereas clone detection algorithms usually compare implementation only. Second, we calculate how many lines are common for a pair of classes, whereas clone detection algorithms have to further analyze which common lines are successive (clones are composed of successive common lines). As a result, the selection algorithm in *GenReferee* is faster, whereas clone detection algorithms are more thorough and expensive. Finally, rough selection in Sect. 5.1 is the first step of *GenReferee* composed of three steps. The results of rough selection would be further analyzed by resolving structural correspondence and computing strength of recommendation. This is one of the reasons why the precision of *GenReferee* is higher than that of *CCFinder*.

Measurement of code similarity is widely explored. It differs from ours in two aspects. First, code similarity measurement usually aims to detect clones (Koschke et al. 2006), retrieve reusable code (Sager et al. 2006), or recommend examples (Holmes et al. 2006), whereas our work aims to recommend generalization refactoring opportunities. Second, code similarity measurement usually compares all methods and attributes of related classes, whereas the proposed approach only focuses on corresponding methods and attributes that may be shared via inheritance.

2.4 Identification of refactoring opportunities

In order to identify refactoring opportunities, experts have summarized typical situations that may require refactoring (Fowler et al. 1999; Wake 2003), and Fowler calls them *Bad Smells* (Fowler et al. 1999, Chap. 3). Researchers have also proposed many detection algorithms to detect bad smells automatically or semiautomatically (Kamiya et al. 2002; Tsantalis and Chatzigeorgiou 2009; Bavota et al. 2011; Bavota et al. 2010; Tsantalis and Chatzigeorgiou 2010; Moha et al. 2006, 2010; Tourwe and Mens 2003; Van Rompaey et al. 2007; Munro 2005).

Travassos et al. (1999) give a set of reading techniques to help developers to identify refactoring opportunities. But the identification is essentially manual. Tourwe and Mens (2003) define detection algorithms with logic rules in SOUL, a logic programming language. Once compiled, the rules are ready to be executed to detect smells. Besides detection, they also gather useful information with the rules to propose refactoring suggestions. Similar work is also reported by Moha et al. (2010, 2006). The difference is that Moha et al. propose a Domain Specific Language (DSL) to specify bad smells, instead of employing an existing logic programming language. The DSL is based on in-depth domain analysis of smells, and thus it is expected to be powerful and convenient. Specifications in DSL are finally transformed into detection algorithms that are ready to be executed. Moha et al. (2010, 2006) also specify all the steps in bad smell specification and detection. Finally, they evaluate the approach on open-source systems. Another category of smell detection algorithms is metric-based. Munro (2005) proposes a metric based approach to detect smells in Java source code. Similar work has been done by Van Rompaey et al. (2007) who propose a metric-based approach to detect two test smells: General Fixture and Eager Test.

Besides the generic detection algorithms and frameworks that in theory are applicable to all kinds of smells, to make smell detection algorithms more efficient people also specially design algorithms for some code smells, e.g., *Clone* (see Sect. 2.3). According to the survey conducted by Zhang et al. (2011), about 54% of papers on code smells focus on code clone, among which 56% focus on clone detection.

Specially designed detection algorithms for other smells other than *Clone* are also available. Tsantalis and Chatzigeorgiou (2009) propose an algorithm to identify *Move Method* refactoring opportunities. Tsantalis and Chatzigeorgiou (2010) propose an algorithm to identify refactoring opportunities introducing polymorphism. Bavota et al. (2010, 2011), propose an algorithm to identify *Extract Class* refactoring opportunities. Tsantalis and Chatzigeorgiou (2011) propose an algorithm to detect long methods and try to decompose them via *Extract Method*. More papers on identification of refactoring opportunities could be found in the survey by Mens and Touwe (2004) and Zhang et al. (2011).

However, these detection algorithms are not suitable for identification of generalization refactoring opportunities. First, the specially-designed detection algorithms discussed above aim at opportunities of other refactorings instead of generalization

```
* Total processing time (ms): 1437 */
/* A is org.\emph{Eclipse}.gef.tools.ConnectionEndpointTracker */
/* B is org.\emph{Eclipse}.gef.tools.SimpleDragTracker */
/* --------------- */
package org.\emph{Eclipse}.gef.tools;
import java.util.List;
........... (omitted)
$commands.ReconnectRequest$UnexecutableCommand;
/* A: import org.\emph{Eclipse}.gef.Tool; */
........... (omitted)
  static final int ;
  /* A: public */
  /* B: protected */
  ConnectionEndpointTracker$SimpleDragTracker$(
  /* A1: ConnectionEditPart cep */
){
    /* A: setConnectionEditPart(cep); */
    /* A: setDisabledCursor(SharedCursors.NO); */
  }
  protected Cursor calculateCursor(){
    if (isInState(STATE$\_$INITIAL | STATE$\_$DRAG |
                  STATE$\_$ACCESSIBLE$\_$DRAG))
      return getDefaultCursor();
    return super.calculateCursor();
  }
........... (omitted)
```

Fig. 1 Sample output of *Breakaway*

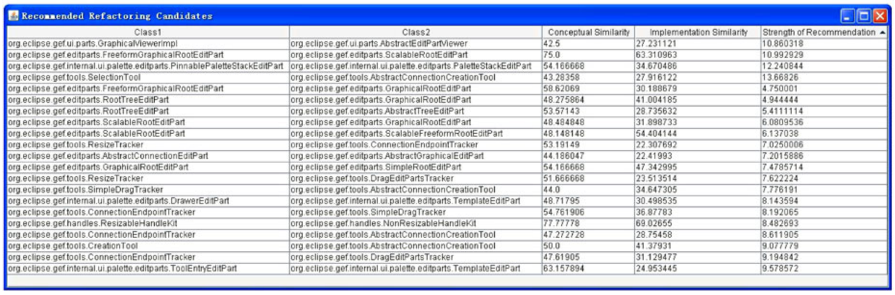
refactoring. Second, the generic detection approaches depend on formal description of refactoring opportunities with predefined specification languages, e.g., SOUL and graph transformation languages. But it is extremely challenging to formalize generalization refactoring opportunities involving method implementation. To the best of our knowledge, no successful cases on its formalization have been publicly reported.

3 Sample scenario

A maintainer realizes (by software metrics or code review) that the inheritance hierarchies in *GEF* [1] is confusing. He decides to restructure the application. The task focuses on correlating classes by inheritance, and sharing common functions among conceptually related classes. These refactorings would make the system easier to read and extend.

For a pair of classes to be generalized, the maintainer detects which methods and attributes are common and how to share them via inheritance. To do so, the maintainer resorts to *Breakaway* (Cottrell et al. 2007) to resolve structural correspondence between classes. The output of *Breakaway* is something like Fig. 1. Statements surrounded by '/*' A and '*/' appear in file A but not in file B. Similarly, statements surrounded by '/* B' and '*/' appear in file B but not in file A. Uncommented statements appear in both files. Some parts of the output are omitted due to space limitation and replaced by '........ *(omitted)'*. As indicated in Fig. 1, *Breakaway* presents detailed differences and commonality between candidate classes. However, which methods or attributes should be shared and how to share them are up to the maintainer. To make

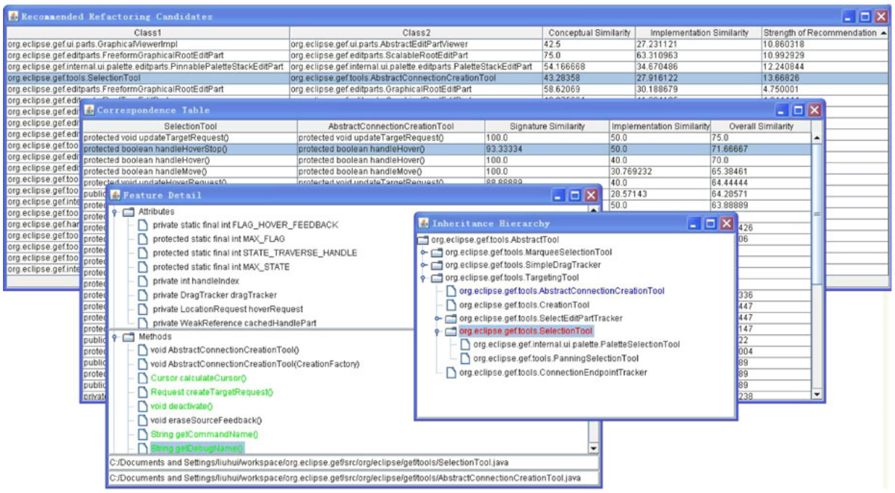
[1] http://www.Eclipse.org/gef/.



Recommended Refactoring Candidates

Class1	Class2	Conceptual Similarity	Implementation Similarity	Strength of Recommendation ▲
org.eclipse.gef.ui.parts.GraphicalViewerImpl	org.eclipse.gef.ui.parts.AbstractEditPartViewer	42.5	27.231121	10.860318
org.eclipse.gef.editparts.FreeformGraphicalRootEditPart	org.eclipse.gef.editparts.ScalableRootEditPart	75.0	63.310963	10.992929
org.eclipse.gef.internal.ui.palette.editparts.PinnablePaletteStackEditPart	org.eclipse.gef.internal.ui.palette.editparts.PaletteStackEditPart	54.166668	34.670486	12.240844
org.eclipse.gef.tools.SelectionTool	org.eclipse.gef.tools.AbstractConnectionCreationTool	43.28358	27.916122	13.66826
org.eclipse.gef.editparts.FreeformGraphicalRootEditPart	org.eclipse.gef.editparts.GraphicalRootEditPart	58.62069	30.188679	4.750001
org.eclipse.gef.editparts.RootTreeEditPart	org.eclipse.gef.editparts.GraphicalRootEditPart	48.275864	41.004185	4.944444
org.eclipse.gef.editparts.RootTreeEditPart	org.eclipse.gef.editparts.AbstractTreeEditPart	53.57143	28.735632	5.4111114
org.eclipse.gef.editparts.ScalableRootEditPart	org.eclipse.gef.editparts.GraphicalRootEditPart	48.484848	31.898733	6.0809536
org.eclipse.gef.editparts.ScalableRootEditPart	org.eclipse.gef.editparts.ScalableFreeformRootEditPart	48.148148	54.404144	6.137038
org.eclipse.gef.tools.ResizeTracker	org.eclipse.gef.tools.ConnectionEndpointTracker	53.19149	22.307692	7.0250006
org.eclipse.gef.editparts.AbstractConnectionEditPart	org.eclipse.gef.editparts.AbstractGraphicalEditPart	44.186047	22.41993	7.2015886
org.eclipse.gef.editparts.GraphicalRootEditPart	org.eclipse.gef.editparts.SimpleRootEditPart	54.166668	47.342995	7.4785714
org.eclipse.gef.tools.ResizeTracker	org.eclipse.gef.tools.DragEditPartsTracker	51.666668	23.513514	7.622224
org.eclipse.gef.tools.SimpleDragTracker	org.eclipse.gef.tools.AbstractConnectionCreationTool	44.0	34.647305	7.776191
org.eclipse.gef.internal.ui.palette.editparts.DrawerEditPart	org.eclipse.gef.internal.ui.palette.editparts.TemplateEditPart	48.71795	30.498535	8.143594
org.eclipse.gef.tools.ConnectionEndpointTracker	org.eclipse.gef.tools.SimpleDragTracker	54.761906	36.87783	8.192065
org.eclipse.gef.handles.ResizableHandleKit	org.eclipse.gef.handles.NonResizableHandleKit	77.77778	69.02655	8.482693
org.eclipse.gef.tools.ConnectionEndpointTracker	org.eclipse.gef.tools.AbstractConnectionCreationTool	47.272728	28.75458	8.611905
org.eclipse.gef.tools.CreationTool	org.eclipse.gef.tools.AbstractConnectionCreationTool	50.0	41.37931	9.077779
org.eclipse.gef.tools.ConnectionEndpointTracker	org.eclipse.gef.tools.DragEditPartsTracker	47.61905	31.129477	9.194842
org.eclipse.gef.internal.ui.palette.editparts.ToolEntryEditPart	org.eclipse.gef.internal.ui.palette.editparts.TemplateEditPart	63.157894	24.953445	9.578572

Fig. 2 Refactoring opportunities recommended by *GenReferee*



Recommended Refactoring Candidates

Class1	Class2	Conceptual Similarity	Implementation Similarity	Strength of Recommendation ▲
org.eclipse.gef.ui.parts.GraphicalViewerImpl	org.eclipse.gef.ui.parts.AbstractEditPartViewer	42.5	27.231121	10.860318
org.eclipse.gef.editparts.FreeformGraphicalRootEditPart	org.eclipse.gef.editparts.ScalableRootEditPart	75.0	63.310963	10.992929
org.eclipse.gef.internal.ui.palette.editparts.PinnablePaletteStackEditPart	org.eclipse.gef.internal.ui.palette.editparts.PaletteStackEditPart	54.166668	34.670486	12.240844
org.eclipse.gef.tools.SelectionTool	org.eclipse.gef.tools.AbstractConnectionCreationTool	43.28358	27.916122	13.66826
org.eclipse.gef.editparts.FreeformGraphicalRootEditPart	org.eclipse.gef.editparts.GraphicalRootEditPart	58.62069	30.188679	4.750001

Correspondence Table

SelectionTool	AbstractConnectionCreationTool	Signature Similarity	Implementation Similarity	Overall Similarity
protected void updateTargetRequest()	protected void updateTargetRequest()	100.0	50.0	75.0
protected boolean handleHoverStop()	protected boolean handleHover()	93.33334	50.0	71.66667
protected boolean handleHover()	protected boolean handleHover()	100.0	40.0	70.0
protected boolean handleMove()	protected boolean handleMove()	100.0	30.769232	65.38461
protected void updateHoverRequest()	protected void updateTargetRequest()	88.88889	40.0	64.44444
public			28.57143	64.28571
protec			50.0	63.88889

Feature Detail

```
Attributes
    private static final int FLAG_HOVER_FEEDBACK
    protected static final int MAX_FLAG
    protected static final int STATE_TRAVERSE_HANDLE
    protected static final int MAX_STATE
    private int handleIndex
    private DragTracker dragTracker
    private LocationRequest hoverRequest
    private WeakReference cachedHandlePart
Methods
    void AbstractConnectionCreationTool()
    void AbstractConnectionCreationTool(CreationFactory)
    Cursor calculateCursor()
    Request createTargetRequest()
    void deactivate()
    void eraseSourceFeedback()
    String getCommandName()
    String getDebugName()
C:/Documents and Settings/liuhui/workspace/org.eclipse.gef/src/org/eclipse/gef/tools/SelectionTool.java
C:/Documents and Settings/liuhui/workspace/org.eclipse.gef/src/org/eclipse/gef/tools/AbstractConnectionCreationTool.java
```

Inheritance Hierarchy

```
org.eclipse.gef.tools.AbstractTool
    org.eclipse.gef.tools.MarqueeSelectionTool
    org.eclipse.gef.tools.SimpleDragTracker
    org.eclipse.gef.tools.TargetingTool
        org.eclipse.gef.tools.AbstractConnectionCreationTool
        org.eclipse.gef.tools.CreationTool
        org.eclipse.gef.tools.SelectEditPartTracker
        org.eclipse.gef.tools.SelectionTool
            org.eclipse.gef.internal.ui.palette.PaletteSelectionTool
            org.eclipse.gef.tools.PanningSelectionTool
        org.eclipse.gef.tools.ConnectionEndpointTracker
```

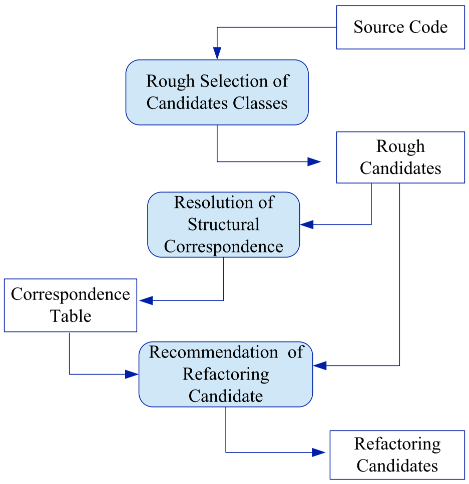
Fig. 3 Detailed information provided by *GenReferee*

such a decision, the maintainer checks the detailed comparison results. He also checks their inheritance hierarchies to make sure that the common methods or attributes can be shared via inheritance because Java does not support multiple inheritance. The inheritance hierarchies also influence the decisions on where the common methods or attributes should be declared. Furthermore, the maintainer checks all methods of the candidate classes and looks them up in ancestors to identify overriding methods that deserve special attention in generalization refactoring.

Because the comparison by *Breakaway* is pairwise, the maintainer must compare $C_{345}^2 = 59,340$ pairs of classes for the 345 classes in *GEF*. However, it is almost impractical to compare $59,340$ pairs of classes with *Breakaway* because intensive human intervention is required in checking the output of *Breakaway* as analyzed in the preceding paragraph.

To ease the work, Cottrell et al. (2007), who proposed *Breakaway*, suggest the maintainer to select a few pairs of candidate classes as input to *Breakaway* with clone detection tools, e.g., *CCFinder* (Kamiya et al. 2002). *CCFinder* reports 138 clone

Fig. 4 Workflow of *GenReferee*


Source Code
Rough Selection of Candidates Classes
Rough Candidates
Resolution of Structural Correspondence
Correspondence Table
Recommendation of Refactoring Candidate
Refactoring Candidates

sets. Although the number of input to *Breakaway* is dramatically reduced, it remains challenging to compare 138 pairs of classes by *Breakaway* because the comparison is tedious and time-consuming. Clone detection tools may miss some refactoring candidates because not all of the candidates share common implementation (clones).

As an alternative, the maintainer tries to identify refactoring opportunities by our tool *GenReferee* instead of clone detection tools. *GenReferee* recommends 21 pairs of classes (presented in Fig. 3) as refactoring opportunities. Although *GenReferee* would compare every pair of classes (totally $C_{345}^{2} = 59{,}340$ pairs) in *GEF* while identifying refactoring opportunities, the comparison is automatic. Consequently, with the help of *GenReferee*, the developer would manually compare 21 pairs of classes only with *Breakaway*.

Each row of the table in Fig. 3 consists a pair of classes on which generalization refactoring may be applied. To inspect the opportunities, the maintainer clicks a row of the table and *GenReferee* presents the following information (shown in Fig. 3): Inheritance hierarchies of the selected classes; Methods and attributes of the selected classes; Detailed structural correspondence between classes; Detailed differences and commonality between classes; Refactoring suggestions.

Detailed comparison of the proposed approach against clone-based approach and fully manual detection is presented in Sect. 7.

4 Overview of the approach

GenReferee proceeds in three phases as depicted in Fig. 4:

(1) *GenReferee* carries out a rough selection of candidate classes on the whole project (source code) with a quick selection algorithm. As a result of the selection, pairwise candidate classes are stored in *Rough Candidates*.
(2) For each pair of candidate classes in *Rough Candidates*, *GenReferee* resolves their structural correspondence. The results are stored in *Correspondence Table* as pairwise methods and attributes.

(3) According to the *Rough Candidates* and *Correspondence Table*, *GenReferee* recommends a list of refactoring opportunities.

The core computation takes place in the last two phases (resolution of structural correspondence and recommendation of refactoring opportunities), whereas the first phase (rough selection of candidate classes) is introduced mainly for performance. The reasons are explained as follows. For a non-trivial project, the number of pairwise combination of classes is huge. The number is $C_N^2 = N \times (N-1)/2$ where N is the total number of existing classes. Moreover, the resolution of structural correspondence between classes (c_i and c_j) is time and resource-consuming (the worst case time complexity is $O(L_1 \times L_2)$ where L_1 and L_2 are length of c_i and c_j, respectively). Consequently, If the resolution of structural correspondence is applied to all of the C_N^2 pairs of classes, *GenReferee* would be too slow (we have tried that by disabling rough selection in Sect. 7.2.2 and it turned out to be almost impractical). To solve this problem, rough selection in the first phase is introduced to select roughly a small set of pairwise classes with a quick selection algorithm, and the selected pairs are checked more rigorously in following phases. As a result, it increases the efficiency of *GenReferee* by shrinking the input to the following phases, whereas the recall and precision of *GenReferee* are not reduced. Moreover, the rough selection also increases the precision of the approach, which has been experimentally validated in Sect. 7.2.2.

5 Preprocessing

5.1 Rough selection

In this phase, *GenReferee* selects candidate classes according to their implementation, as well as terms in their declarations.

Inheritance correlates classes belonging to the same conceptual category to share common interfaces and common source code. If two similar classes share some duplicate or similar source code but there is no conceptual relationship between them, delegation, instead of inheritance, is the right mechanism to remove duplicate code (Kegel and Steimann 2008). Otherwise, the inheritance hierarchy would be confusing because the hierarchy in the program would differ from what it is in the real world.

GenReferee selects conceptually related classes according to terms in their declarations, i.e., nouns and verbs extracted from class names, attribute names, method names, and parameter names. Currently *GenReferee* extracts words according to capitalized letters assuming the names follow the widely used camel-case or Pascal-case capitalization styles.

Similarity in implementation is another criterion for candidate selection. This is one of the reasons why (Cottrell et al. 2007) suggested to select candidate classes by clone detection tools. Term-based selection might be applied prior to implementation-based selection because the former is faster and it might filter much more false candidates as suggested by our evaluation in Sect. 7.2. Changing the application order might influence performance, but it would not influence detection results.

```
/*Input:
 Array SimArray (Similarity between pairwise methods);
 MSM (Minimal similarity between corresponding methods)
*/
Sort SimArray in descending order;
Find location ENDPOSITION where
      SimArray[ENDPOSITION].Similarity < MSM and
      SimArray[ENDPOSITION-1].Similarity >= MSM
if (SimArray[0]<MSM)
    ENDPOSITION=0
if (SimArray[Sm.Length-1]>=MSM)
    ENDPOSITON=Sm.Length
Method a,b;
for (i=0;i < ENDPOSITION;i++)
{// the pair of similar methods
     a=SimArray[i].Method1;
     b=SimArray[i].Method2;
   if (a.Corresp==null and b.Corresp==null)
       //neither of them has correspondence yet
      {    //make them correspond to each other
           a.Corresp=b;
           b.Corresp=a;
      }
}
```

Fig. 5 Correspondence resolution algorithm

5.2 Resolution of structural correspondence

For each pair of classes selected in the preceding phase, *GenReferee* in this phase resolves their structural correspondence. The structural correspondence is represented as correspondence tables like that in Fig. 3.

Currently, *GenReferee* resolves correspondence between attributes according to both types and names. Two attributes correspond if and only if they are of the same type and share the same name. Type comparison of attribute types has not considered type hierarchies, i.e., inheritance between different types. The same is true for method comparison.

Resolving correspondence between methods is more complex than that of attributes. First, we define the overall similarity between two methods as a synthesis of the similarities in signature and implementation:

$$S_m(m_i, m_j) = \frac{1}{2} \times \left[S_{sg}(m_i, m_j) + S_{imp}(m_i, m_j)\right] \qquad (1)$$

where $S_m(m_i, m_j)$ is the overall similarity between methods m_i and m_j. S_{sg} and S_{imp} are similarities in signature and implementation, respectively. The coefficient $1/2$ makes S_m range between 0 and 1.

With the similarity S_m, we propose an algorithm in Fig. 5 to resolve the correspondence between two sets of methods. In the algorithm, every pair of similar methods is checked one by one to determine whether they correspond or not. The algorithm guarantees that the most similar methods correspond to each other by checking the most similar pairs first (line 5 sorts *SimArray* in descending order). It also guarantees that each method corresponds to no more than one method by the *if-condition* in line 18. The complexity of the algorithm is $O(n \log n)$ where n is the size of array *SimArray*.

If overriding methods of classes c_i and c_j have the same signature, and the signature has been declared in a common ancestor of c_i and c_j, the overriding methods are deemed to correspond though their signatures might not be identical (return type of the overridden method might be refined by overriding methods). In this case, we do not even have to compute their similarity at all.

A special case is that there is a direct inheritance between classes c_i and c_j, e.g., c_i is a direct ancestor of c_j. In this case, *GenReferee* compares all methods of the child (e.g., c_j) to those of the ancestor (c_i): two methods correspond if they have the same signature.

6 Recommendation

With the correspondence table *Crs* generated in Sect. 5.2, *GenReferee* computes the strength of recommendation for each pair of classes selected in Sect. 5.1, and recommend a list of refactoring opportunities.

6.1 Strength of recommendation

A refactoring candidate (c_i, c_j) is composed of a pair of similar classes c_i and c_j. According to their positions in inheritance hierarchies, candidate pairs are classified into the following four categories.

The first category is *root classes*, i.e., neither c_i nor c_j has superclass other than $Object$ that is the default ancestor of all Java classes. In other words, c_i and c_j are root classes. In this case, their common functions, both interfaces and implementation, can be shared by introducing a new superclass. The strength of recommendation is the sum of similarities between corresponding methods and attributes:

$$R(c_i, c_j) = \sum_{(m_i, m_j) \in Crs} S_m(m_i, m_j) + \sum_{(a_i, a_j) \in Crs} S_a(a_i, a_j) \tag{2}$$

where $R(c_i, c_j)$ is the strength of recommendation, *Crs* is the correspondence table generated in Sect. 5.2, m_i and m_j are methods of classes c_i and c_j, respectively. a_i and a_j are attributes of classes c_i and c_j, respectively. S_m and S_a are similarities between corresponding methods and attributes, respectively.

The second category is *sibling classes*, i.e., classes c_i and c_j directly inherit from the same superclass c_{sup}. Their common functions can be pulled up to their superclass c_{sup} if these functions are also owned by other children of c_{sup}. The strength of recommendation is computed as follows:

$$\begin{aligned} R(c_i, c_j) = {} & \big(|Children(c_{sup})| - 1\big) \\ & \times \Bigg(\sum_{(m_i, m_j) \in Crs'} S_m(m_i, m_j) + \sum_{(a_i, a_j) \in Crs'} S_a(a_i, a_j) \Bigg) \end{aligned} \tag{3}$$

where Crs' is a subset of Crs. Every method or attribute in Crs' has correspondence in every child of c_{sup}. $|Children(c_{sup})|$ is the total number of the direct children of c_{sup}. To calculate Crs', we should search every subclass of c_{sup} for correspondence of every common method or attribute between classes c_i and c_j.

The coefficient $(|Children(c_{sup})| - 1)$ makes sure that if classes c_i and c_j are the only two subclasses of c_{sup}, the resulting $R(c_i, c_j)$ of Formula (3) is equal to that computed according to Formual (2).

As an alternative, it is also possible to extract the common functions as a new class inheriting from c_{sup}, and then c_i and c_j inherit from the new class. In this case, the strength of recommendation should be computed according to (2). The two alternative solutions are compared by their strength of recommendation and the one with greater strength is recommended.

The third category is *unrelated classes*, i.e., c_i and c_j have different superclasses (not *Object*), and they have no common ancestor besides *Object*. As a result, only interfaces can be shared because Java does not support multiple inheritance. The strength of recommendation should be computed according to (4).

The fourth category is *offspring of the same ancestor*, i.e., classes c_i and c_j indirectly inherit from the same ancestor c_{anc} (not *Object*), but they have different direct super-classes. In this case, their common methods (method signatures only) must be extracted as an interface instead of a common superclass because Java does not support multiple inheritance. In this case, the strength of recommendation is computed as follows:

$$R(c_i, c_j) = \frac{1}{2} \times \sum_{(m_i, m_j) \in Crs} S_{sg}(m_i, m_j) \tag{4}$$

where S_{sg} is the signature similarity between corresponding methods. In (4), the strength of recommendation is the sum of similarities between corresponding method signatures. Similarities in implementation and attributes are not accounted for because implementation and attributes cannot be extracted to interfaces. The coefficient $\frac{1}{2}$ comes from (1) (by setting $S_{imp}(m_i, m_j) = 0$).

As an alternative to extracting common interface, it is also possible to pull up the common functions to the common ancestor c_{anc} if most common functions are shared by all offspring of c_{anc}. In this case the strength of recommendation is computed by (3), where Crs' is a collection of methods and attributes owned by all offspring of c_{anc}. The two alternative solutions are compared by their strength of recommendation and the one with greater strength is recommended.

6.2 Recommended list

With the strength of recommendation of each refactoring opportunity, *GenReferee* recommends those whose strength of recommendation is higher than a predefined threshold, *Minimal Strength of Recommendation* (*MSR*). The resultant recommended list is something like that in Fig. 2. Software engineers may check them manually one after the other and make the final decision.

For each recommended refactoring opportunity, *GenReferee* also generates refactoring suggestions: which method and attribute should be shared via inheritance and how to share them. The suggestions are generated based on the classification and analysis in Sect. 6.1.

6.3 Grouping

It is also possible to share common functions among a group of classes. For example, refactoring opportunities $\langle c_1, c_2\rangle$, $\langle c_2, c_3\rangle$, and $\langle c_1, c_3\rangle$ indicate that some common functions might be shared among c_1, c_2, and c_3. We present classes as nodes of an undirected graph, and connect c_i and c_j if $\langle c_i, c_j\rangle$ is recommended by *GenReferee* as a refactoring opportunity. As a result, the task to detect generalization opportunities involving more than two classes is transformed into the identification of maximal complete undirected subgraphs, known as the *Clique Problem*. Although the *Clique Problem* in general is NP-complete, the grouping is fast because the graph is usually small and sparse. For example, the graph produced on *JFace* (a middle-sized application containing 332 classes, and more than 40 thousand lines of source code) has only 16 vertexes and 8 edges. In our experiments conducted on five non-trivial applications (Sect. 7), 31 refactoring opportunities were detected, but only one group consisted of three classes was reported.

7 Evaluation

7.1 Experimental setup

The experiment investigates the following questions.

(1) How to set thresholds of the proposed approach to achieve a good performance?
(2) How well does the approach perform compared to existing approaches (clone-based detection and fully manual detection)?

As discussed in Sect. 1, generalization refactoring opportunities are currently identified manually by software engineers or semi-automatically with clone detection tools (clone-based detection) (Cottrell et al. 2007). Section 7.3 compares the proposed approach against clone-based detection with *CCFinder* (Kamiya et al. 2002) or *Aries* (Higo et al. 2004). Section 7.4 compares the proposed approach against manual identification.

The experiment was conducted on five applications: *GEF*,[2] *Java Source Metric (JSM)*,[3] *JFace*,[4] *Thout Reader*,[5] and *AutoMeD*. A brief summary of the subject applications is presented in Table 1.

GEF is a framework for development of graphical editors in which blocks and lines can be dragged and dropped. The framework (version 3.4) is composed of 324 files, 344 classes, and 27,224 lines of source code. *Java Source Metric* was designed

[2]http://www.Eclipse.org/gef/.

[3]http://www.trustie.net/projects/files/list/PKUCodeMetric.

[4]*Eclipse* CVS:pserver:anonymous@dev.*Eclipse*.org:/cvsroot/*Eclipse*.

[5]http://thout.sourceforge.net/.

Table 1 Subject applications

	GEF	JSM	JFace	Thout Reader	AutoMeD
Open-source	Yes	Yes	Yes	Yes	No
Size (LOC)	27,224	3,693	41,836	19,377	5,345
Number of classes	344	60	332	269	40
Depth of inheritance tree	3.105	2	2.154	2.755	1.65
Version	3.4	1.4.2	3.4	1.8.0	1.0
Domain	GUI framework	Metrics	UI toolkit	Document reader	Refactoring tool

to measure metrics of the java source code. Version 1.4.2 contains 60 classes and 3,695 lines of source code. *JFace* is a toolkit with solutions of common UI programming tasks. The toolkit (version 3.4) is composed of 379 files consisting of 332 classes and 41,836 lines of source code. *Thout Reader* is a document manager, allowing users to browse, search, bookmark, and append documents. *Thout Reader* (version 1.8.0) is composed of 269 classes, and 19,377 lines of source code. *AutoMeD* was developed by students of the authors to generate refactoring suggestions. *AutoMeD* (version 1.0) contains 40 classes and 5345 lines of java source code.

The first four applications were selected for the following reasons. First, they are open-source applications, which makes it possible for other researches to replicate the experiment. Second, they are non-trivial, and contain complex inheritance hierarchies to be improved by generalization refactoring (according to our initial informal analysis). Finally, they are from different domains and developed by different developers. Carrying out evaluation on these applications may help generalize the conclusions. The last application, *AutoMeD*, was selected because it is small-sized, and the experimenters were familiar with it, which makes it suitable for manually checking. Fully manual identification of generalization opportunities in large unfamiliar applications is challenging.

Seven postgraduates majoring in software engineering were selected to participate the experiment. All of them had rich experience in software refactoring, and some experience with *CCFinder*. All of them had more than three years of experience with Java and *Eclipse*.

Quantitative evaluation of the approaches was conducted by *precision* and *recall*:

$$\begin{cases} Precision = \frac{Num.\ of\ True\ Opportunities\ Recommended}{Num.\ of\ All\ Opportunities\ Recommended} \\ Recall = \frac{Num.\ of\ True\ Opportunities\ Recommended}{Num.\ of\ True\ Opportunities} \end{cases} \tag{5}$$

As discussed above, manual identification of generalization refactoring opportunities in large unfamiliar applications is extremely challenging. Consequently, recall of the approach is hard to calculated because it depends on the number of true opportunities that in turn might require manual identification. As a result, we would not calculate its recall on large applications, e.g., *GEF* and *JFace*, but we might calculate it on smaller applications, e.g., *Thout Reader* and *AutoMeD*.

7.2 Calibration on GEF and JSM

7.2.1 Determining thresholds

Thresholds of the approach include: *MSM* (Minimal Similarity between Methods), *MSR* (Minimal Strength of Recommendation for refactoring opportunities), *MSC* (Minimal Strength of Conceptual relationship between classes), and *MIS* (Minimal Implementation Similarity between classes).

First, we conservatively set all thresholds to discover as many refactoring opportunities as possible. The initial values of *MSR*, *MSM*, *MSC* and *MIS* were at first set to 1.5, 0.1, 0.2 and 0.1, respectively. With the initial setting, the approach was applied to *GEF* and it recommended 30 opportunities automatically. The two selected postgraduates manually checked all of the recommended opportunities with the help of *Eclipse* IDE, *UltraCompare* and *Breakaway*. Finally they identified 14 true opportunities (refactorable). The precision is $14/30 = 46.67\%$. The same experiment was also done on *JSM*, and the precision is $2/9 = 22.22\%$.

For the 16 (14 in *GEF* and 2 in *JSM*) pairs of classes confirmed manually (true opportunities recommended), the minimal strength of recommendation was 4.7, the minimal strength of conceptual relationship was 0.44, and the minimal implementation similarity was 0.22. We also calculated the similarity between corresponding methods of the 16 pairs of classes, and the minimal similarity was 0.16. Therefore, increasing *MSR*, *MSM*, *MSC* and *MIS* from the initial values (1.5, 0.1, 0.2 and 0.1, respectively) to 4.7, 0.16, 0.44, 0.22 respectively would not cause *GenReferee* to miss any of the 16 true opportunities, whereas the precision would be improved because false opportunities would be eliminated by the increased thresholds. With this setting, we reapplied *GenReferee* to *GEF* and it recommended 21 opportunities. The precision was improved from $14/30 = 46.67\%$ (with the initial setting) to $14/21 = 66.67\%$ (with increased thresholds). With the new setting, the precision on *JSM* was also improved from $2/9 = 22.22\%$ to $2/4 = 50\%$.

7.2.2 Effect of different phases

With the setting obtained in Sect. 7.2.1 ($MSM = 0.16$, $MSR = 4.7$, $MIC = 0.44$, and $MIS = 0.22$), we would evaluate the effect of different phases of the proposed approach in this section.

To evaluate the effect of the last two phases of the approach (*resolution of structural correspondence* and *recommendation of refactoring candidates*), we disabled the last two phases by setting $MSR = 0$. With this setting, the rough selection in the first phase worked alone, and it recommended 112 generalization opportunities in GEF. Among them, 14 opportunities were confirmed to be refactorable. The precision is $14/112 = 12.50\%$, much lower than that (66.67%) when the last two phases are employed. The difference in precision suggests that the last two phases have a strong effect on the precision of the approach. It also suggests that the rough selection alone is not accurate.

To evaluate the effect of term-based selection, we disabled the term-based selection by setting $MSC = 0$, and reapplied *GenReferee* to GEF. After 513.52 seconds,

Table 2 Effect of different phases

	Test 1	Test 2	Test 3	Test 4	Test 5
Term based selection	Enabled	Enabled	Enabled	Disabled	Disabled
Implementation based selection	Enabled	Enabled	Disabled	Enabled	Disabled
The last two phases	Enabled	Disabled	Enabled	Enabled	Enabled
Opportunities recommended	21	112	24	463	–
True opportunities recommended	14	14	14	14	–
Precision	66.67%	12.50%	58.33%	3.02%	–
Running time (second)	5.66	4.3	6.16	513.52	>15 hours

GenReferee recommended 463 generalization opportunities. Among them, 14 opportunities were confirmed by two independent engineers to be refactorable. The precision is $14/463 = 3.02\%$. This is expensive and inaccurate compared to the default setting in which term-based selection is employed. The default setting achieved a precision of 66.67% in 5.66 seconds. The results of the comparison suggest that term-based selection has a strong effect on the precision and efficiency of the approach.

To evaluate the effect of implementation-based selection, we disabled the implementation based selection by setting $MIS = 0$. With this setting, we reapplied *GenReferee* to GEF, and it recommended 24 generalization opportunities after 6.16 seconds. Among them, 14 opportunities were confirmed to be refactorable. The precision ($14/24 = 58.33\%$) is close to that (66.67%) in the default setting in which implementation-based selection is employed. The precision is much higher than that (3.02%) when term-based selection is disabled. The results suggest that implementation-based selection has a weaker effect on precision and efficiency than term-based selection. However, if the threshold of implementation similarity is set too high, the recall of the approach would be reduced dramatically. For example, if the threshold *MIS* increases from 0.22 to 0.4, 9 out of 14 true opportunities ($9/14 = 64.29\%$) would be missed. It is consistent with our expectation that certain generalization opportunities contain few common statements.

To evaluate the necessity of rough selection, we disabled the rough selection (including both term-based selection and implementation-based selection) by setting $MSC = MIS = 0$. With this setting, we reapplied *GenReferee* to GEF. After a whole night (more than 15 hours), it was still running and no recommendation was generated yet (environment: Window XP, Intel Core2 3.0 GHz, 2 GB memory, 250 GB hard disk). The results suggest that *GenReferee* without rough selection is almost impractical.

The evaluation results are summarized in Table 2. Columns 2–6 present the settings and results of the five experiments.

7.3 Comparison against clone-based approaches

In this section, we compare the proposed approach against two clone-based approaches on *JFase* and *Thout Reader*.

It should be noted that the experiment is not to compare *GenReferee* and clone detection tools. The comparison would be nonsensical because clone detection tools are not specifically designed to detect generalization refactoring opportunities. What the experiment tries to compare is two alternative approaches to identify generalization refactoring opportunities: the proposed approach, and the clone-based identification.

As discussed in Sect. 1, currently developers lack efficient tools support specially designed for the identification of generalization refactoring opportunities. Consequently, developers have to do it manually or resort to clone detection tools though these tools are not specially designed for this purpose.

One of the compared clone-based approaches employs *CCFinder*. *CCFinder* is a well-known clone detection tool, and it has been employed to identify generalization refactoring opportunities (Cottrell et al. 2007). The other clone-based approach compared in this section employs *Aries* (Higo et al. 2004). *Aries*,[6] as a part of ICCA (Integrated Code Clone Analyzer),[7] is a *CCFinder* based refactoring assistant. It analyzes detection results of *CCFinder*, and then calculates metrics to support refactoring activities. It also filters clone sets by these metrics. With these metrics, it might distinguish clone pairs suitable for generalization refactoring (*Pull up Method*) from others that should be resolved by *Extract Method*.

7.3.1 Results

GenReferee was applied to *JFace*, and it recommended 17 opportunities. An experimenter manually checked these opportunities. He was allowed to use Eclipse IDE, *Breakaway*, and *UltraCompare*[8] to facilitate the checking. The manual checking took nearly 5 hours, and he confirmed 8 opportunities to be refactorable. Results are presented in Table 3 (columns 1–3). The precision of the recommendation is $8/17 = 47\%$.

Another experimenter identified refactoring opportunities with the help of *CCFinder*. To make the comparison fair, he calibrated *CCFinder* on *GEF* and *JSM* first. The main threshold of *CCFinder* is the *Minimum Clone Length*. With increased *Minimum Clone Length*, *CCFinder* would report fewer clones and thus improve its precision, whereas its recall might decrease. The calibration is done favoring recall (i.e., getting the largest *Minimum Clone Length* which would not miss any of the found generalization refactoring opportunities), as *GenReferee* has done in Sect. 7.2. As a result, the *Minimum Clone Length* of *CCFinder* was set to 60. With this setting, *CCFinder* selected 148 clone sets from *JFace* for further manual analysis. For each clone set, the experimenter checked their enclosing classes to determine whether they are good generalization opportunities. He was also allowed to use *Breakaway*, Eclipse IDE and *UltraCompare*. The checking took him 18 hours, and he confirmed 6 pairs of classes to be refactorable by generalization refactoring. Results are presented in Table 3 (columns 1, 2, and 4). The precision is $6/148 = 4.05\%$.

Another experimenter identified refactoring opportunities with the help of *Aries*. *Aries* reported 56 clone pairs, but only 9 of them have a non-zero *DCH* (*Dispersion of Class Hierarchy*) (Higo et al. 2004). According to the definition of *DCH* (Higo et al.

[6]http://sel.ist.osaka-u.ac.jp/icca/aries-e.html.

[7]http://sel.ist.osaka-u.ac.jp/icca/index-e.html.

[8]http://www.ultraedit.com/products/ultracompare.html.

Table 3 Evaluation results on JFace

Class pairs	*GenReferee*	*CCFinder*	*Aries*
viewers.TableViewerRow viewers.TreeViewerRow	Yes	Yes	Yes
AbstractTableViewer viewers.AbstractTreeViewer	Yes	No	No
viewers.AbstractListViewer viewers.AbstractTableViewer	Yes	No	No
viewers.ViewerCell viewers.ViewerRow	Yes	Yes	No
viewers.CheckboxTableViewer viewers.CheckboxTreeViewer	Yes	Yes	No
layout.GridDataFactory layout.RowDataFactory	Yes	Yes	No
viewers.ComboBoxCellEditor viewers.ComboBoxViewerCellEditor	Yes	Yes	No
viewers.TreeViewerEditor viewers.TableViewerEditor	Yes	Yes	No
Total Effort ($man \cdot hour$)	5	18	2.5
Precision (*true opportunities recommended* ÷ *all candidates recommended*)	47%	4.05%	11.1%

The same prefix (org.*Eclipse*.jface.) of all qualified class names is omitted due to space limitations

2004), if the *DCH* of a clone pair is zero, its duplicate fragments belong to the same class. In this case, *Extract Method*, instead of *Pull up Method*, should be applied. Consequently, to identify generalization refactoring opportunities, clone pairs whose *DCH* are zero could be ignored. It took the experimenter about 2.5 hours to analyze the 9 clone pairs whose *DCH* is non-zero. He found that only one pair of them might benefit from generalization refactoring. The precision was $1/9 = 11.1\%$.

As a further evaluation, the comparison was also done on *Thout Reader*. The process was similar to that on *JFace*, but two experimenters exchanged their roles in this part by exchanging their approaches. The experimenter checking *JFace* with *GenReferee* was assigned to check *Thout Reader* with *CCFinder*. Simultaneously, the other experimenter previously checking *JFace*with *CCFinder* was assigned to check *Thout Reader* with *GenReferee*. The exchange helps to reduce bias. A new experimenter was assigned to check *Thout Reader* with *Aries*.

Evaluation results are presented in Table. 4. On *Thout Reader*, *GenReferee* recommended 7 generalization opportunities. The appointed experimenter spent 1.5 hours in checking them, and 5 of them were confirmed to be refactorable. The precision is $5/7 = 71.43\%$. *CCFinder* reported 70 clone sets, and the appointed experimenter spent 5 hours in checking the clone sets, finding 3 pair of classes to be true generalization opportunities. The precision is $3/70 = 4.29\%$. *Aries* reported 35 clone pairs among which 25 have non-zero *DCH*. After 3 hours' manual checking, the experimenter found one of them might benefit from generalization refactoring. Consequently, the precision is $1/25 = 4\%$.

Table 4 Evaluation results on thout reader

Class pairs	GenReferee	CCFinder	Aries
viewer.library.LibraryJTreePoupMouseListenerImpl viewer.bookmarks.BookmarkJTreePopupMouseListenerImpl	Yes	Yes	Yes
utils.ThoutEncryptedZipEntryPackageWrapter utils.ThoutZipEntryPackageWrapper	Yes	Yes	No
viewer.browse.BrowseJTreePopupMouseListenerImpl viewer.bookmarks.BookmarkJTreePopupMouseListenerImpl	Yes	No	No
viewer.library.LibraryJTreePopupMouseListenerImple viewer.browse.BrowseJTreePopupMouseListenerImpl	Yes	No	No
viewer.search.quick.SearchPanePopupMouseListener viewer.menu.JPopupMouseListenerImpl	Yes	Yes	No
Total effort ($man \cdot hour$)	1.5	5	3
Precision	71.43%	4.29%	4%

The same prefix (osoft.) of all qualified class names is omitted due to space limitations

7.3.2 Analysis

Compared to *CCFinder*-based detection, the proposed approach achieved a significant reduction in effort ranging from $(5 - 1.5)/5 = 70\%$ (on *Thout Reader*) to $(18 - 5)/18 = 72.22\%$ (on *JFace*). One of the possible reasons for the significant reduction in effort is that *GenReferee* has a much higher precision. The precision of *GenReferee* is 47% and 71.43% on *JFace* and *Thout Reader*, respectively. However, the precision of *CCFinder* is 4.05% and 4.29% on *JFace* and *Thout Reader*, respectively. The results are consistent with our expectation. Recall of *CCFinder* is low because *CCFinder* was not especially designed for this task. To the best of our knowledge, no existing tools have been specially designed for this purpose. This is the reason why we especially propose an approach in this paper for this issue.

From Tables 3 and 4, we observed that *GenReferee* achieved a higher recall : all true opportunities detected by *CCFinder* were also detected by *GenReferee*, but the reverse is not true. 4 out of 13 true opportunities detected by *GenReferee* were missed by *CCFinder*. Concrete recall cannot be calculated because there is no guarantee that all generalization opportunities have been detected by *GenReferee* or *CCFinder*. But quantitative comparison of recall is practical because *GenReferee* and *CCFinder* were applied to the same target applications. If the total number of true opportunities in the target applications is N, the recall of *GenReferee* and *CCFinder* are $13/N$ and $9/N$, respectively. Therefore, the recall of *GenReferee* is $\frac{13/N}{9/N} = 1.44$ times of that of *CCFinder*.

CCFinder missed the 4 pairs of candidates for the following two reasons. First, some common methods to be extracted are simple, and their common implementation is too short to be reported as clones. Refactoring candidate (*viewer.library.-LibraryJTreePopupMouseListenerImple*, *viewer.browse.BrowseJTreePopupMouse-ListenerImpl*) in *Thout Reader* is a good example. Their common methods, *mouse-Clicked*, *mouseEntered*, *mouseExisted*, *getRootFolderPopupMenu*, *getEmptyFolder-PopupMenu*, *getFolderPopupMenu*, and *getNodePopupMenu*, should be pulled up to their common superclass *JTreePopupMouseListenerImpl*. These methods are simple and short. For example, method *mouseClicked* contains only 3 lines of source code. As a result, *CCFinder* did not find any clone between the two classes because clone detection algorithms usually employ a threshold (minimal length of clones) to exclude small clones. Extracting and sharing small clones by *delegation* is rarely worthwhile. But sharing common short methods by *inheritance*, as what happened in the evaluation, is an entirely different matter.

Second, corresponding methods to be extracted as common interfaces usually have different implementation. As a result, they do not contain any clone. Refactoring candidate (*org.Eclipse.jface.viewers.AbstractListViewer*, *org.Eclipse.jface.viewers.AbstractTableViewer*) in *JFace* is a good example. The two classes have the same ancestor *StructuredViewer*, and some methods, e.g., *add(Object)*, *add(Object[])*, *remove(Object[])*, *remove(Object)*, *insert(Object,int)* and *getElementAt(int)*, are defined in both of them (but not in other offspring of *StructuredViewer*). Furthermore, some common methods, e.g., *add(Object[])*, have different implementation in different classes. Consequently, the experimenter decided to pull them up to a new interface, and force *AbstractListViewer* and *AbstractTableViewer* to implement the new interface. *CCFinder* reported no duplicate code between *AbstractListViewer* and *AbstractTableViewer* because these common methods are either too short to be reported as clones, e.g., *add(Object)*, or implemented differently, e.g., *add(Object[])*.

GenReferee did not miss the 4 pairs of classes although it also requires classes to be similar in implementation. First, the threshold of implementation similarity is low, and thus easy to meet. Low threshold usually leads to low precision. But *GenReferee* employs other techniques (such as term-based selection and filtering through strength of recommendation) to improve its precision. Second, *GenReferee* requires classes to share some common statements instead of clones (only sizable fragments composed of consecutive common statements can be called clones). Short common methods usually contain some common statements though they may not contain clones.

As expected, the recall of *Aries* was not higher than that of *CCFinder*. Compared to *CCFinder* and *GenReferee*, its recall was reduced by $(9-2)/9 = 77.8\%$ and $(13-2)/13 = 84.62\%$, respectively. One of the reasons is that *Aries* was based on detection results of *CCFinder*. Consequently, refactoring opportunities missed by *CCFinder* would not be identified by *Aries*. Another possible reason is that *Aries* filtered clone pairs with metrics, and thus it might miss some true opportunities reported by *CCFinder*.

We had expected *Aries* to achieve much higher precision than *CCFinder* because of the following two reasons. First, it would apply further analysis on results of *CCFinder*, and remove those not suitable for refactoring. Second, it would classify clone pairs into two categories by *DCH*: those for *Pull up Method* and others for *Extract Method*. However, it turned out that the precision improvement was not as great as expected (the precision was even reduced on project *Thout Reader*). One of the possible reasons is that some true opportunities had been filtered by *Aries*: *CCFinder* had reported 9 true opportunities, whereas only 2 had been found by *Aries*. In other words, 7 out of 9 ($7/9 = 77.8\%$) true opportunities had been filtered by *Aries*.

Table 5 Comparison against Manual Identification on AutoMeD

Refactoring opportunities	*GenReferee*	Manual identification
MethodCompareInfo MethodInfo	Yes	Yes
DBLogial ASTUtils	Yes	Yes
Total Effort ($man \cdot hour$)	0.2	9
Precision	100%	–
Recall[‡]	100%	–

‡ Assuming all true opportunities had been identified by the participants

7.4 Comparison against manual identification

In this section, we compare the proposed approach with manual identification. The evaluation was carried out on *AutoMeD* and *Thout Reader*. Manually checking large unfamiliar applications is challenging. That is why the evaluation was carried out on *AutoMeD*: it is middle-sized, and the experimenters were familiar with it. *Thout Reader* is an open-source larger application, and evaluation on this application would reduce threats to validity.

Evaluation results on *AutoMeD* are presented in Table 5. Experimenters working with and without *Genreferee* reported the same results: two generalization opportunities were detected. However, the identification with *GenReferee* took 0.2 man-hour only, whereas manual identification took 9 man-hours. Compared to manual identification, *GenReferee* reduced checking effort by $(9 - 0.2)/9 = 97.8\%$. Based on the assumption that all refactoring opportunities had been identified by manual identification, the recall of *Genreferee* was $2/2 = 100\%$.

To make conclusions more reliable, the comparison between the proposed approach and fully manual identification was also carried out on an open-source project *Thout Reader*. Identification of generalization refactoring opportunities in *Thout Reader* with *GenReferee* has already been done and reported in Section 7.3. For the comparison, another three engineers manually identified generalization refactoring opportunities in *Thout Reader*. They first worked independently, and reported a set of refactoring candidates. After that, they would discuss together to give a full list of potential refactoring opportunities. We asked three (instead of only one) engineers to do manual identification because it was expected to be challenging to manually identify all potential refactoring opportunities in a unfamiliar and non-trivial application containing nearly twenty thousand lines of source code and 269 classes. Assigning more than one participators would help to find more potential refactoring opportunities. It, in turn, might make the recall of *GenReferee* more reliable because the recall was calculated based on the total number of refactoring opportunities manually confirmed.

Evaluation results are presented in Table 6. Rows 3–5 present the results of manual identification. E_1, E_2, and E_3 represent the three engineers involved in manual identification.

Table 6 Comparison against manual identification on thout reader[†]

Refactoring opportunities	*GenReferee*	Manual ientification		
		E_1	E_2	E_3
viewer.library.LibraryJTreePoupMouseListenerImpl viewer.bookmarks.BookmarkJTreePopupMouseListenerImpl	Yes	Yes	Yes	Yes
utils.ThoutEncryptedZipEntryPackageWrapter utils.ThoutZipEntryPackageWrapper	Yes	Yes	No	Yes
viewer.browse.BrowseJTreePopupMouseListenerImpl viewer.bookmarks.BookmarkJTreePopupMouseListenerImpl	Yes	Yes	Yes	Yes
viewer.library.LibraryJTreePopupMouseListenerImple viewer.browse.BrowseJTreePopupMouseListenerImpl	Yes	Yes	Yes	Yes
viewer.search.quick.SearchPanePopupMouseListener viewer.menu.JPopupMouseListenerImpl	Yes	Yes	Yes	Yes
viewer.Vista viewer.JComponentVista	No	Yes	Yes	No
Total effort ($man \cdot hour$)	1.5	60	45	40
Precision	71.43%	–	–	–
Recall[‡]	83.3%	100%	83.3%	83.3%

[†]The same prefix (osoft.) of all qualified class names is omitted due to space limitations

[‡]Recall was computed with the assumption that all true opportunities had been identified by the participants

From the table, we observe that two of the three engineers manually identifying refactoring opportunities had missed some potential refactoring opportunities. It consists with our expectation that manual identification is challenging. Compared to *GenReferee*, manual identification was time-consuming. On average, manual identification cost 48 man hours, whereas identification with *GenReferee* cost 1.5 man hours only.

Assuming all potential refactoring opportunities had been identified by these engineers, the recall of *GenReferee* was $5/6 = 83.3\%$, and the average recall of manual identification was $(6 + 5 + 5)/(6 \times 3) = 88.9\%$. The only one missed by *GenReferee* involved two classes: *osoft.viewer.Vista* and *osoft.viewer.JComponentVista*. The later is the only child of the former. Because there is no direct reference to the superclass *osoft.viewer.Vista*, engineers decided that the two classes should be combined. They pulled up all methods and attributes of the child to its superclass, and retyped all of its references.[9] *GenReferee* missed this refactoring opportunity because there is no common implementation or common interfaces to be shared among classes.

The comparison on *AutoMeD* was done by engineers familiar with the application, whereas comparison on *Thout Reader* was done by engineers unfamiliar with the evaluation subject. The experimenters were familiar with *AutoMeD* because they had been involved in the development of *AutoMeD*. But *Thout Reader* was an open-source application that the experimenters had never known before the evaluation.

[9]It is also possible to pull down all methods and attributes from the superclass, and remove the superclass. In this case, no generalization refactoring would be applied.

However, the comparison results on the two applications were consistent. On both of the applications, recall of *GenReferee* was close to that of manual identification, whereas working effort was reduced by more than 90%.

7.5 Threats to validity

As discussed in Sect. 1, generalization opportunities are currently detected manually by software engineers or semi-automatically by clone detection tools. The experiment compared the proposed approach with manual identification against clone-based approaches where *CCFinder* and *Aries*. But other clone detection tools, e.g., *CloneDR*, might be used to do that, too. The experiment has not compared these potential approaches and thus it might be a threat to external validity. We selected *CCFinder* because of the following two reasons. First, it was recommended by experts (Cottrell et al. 2007) to detect generalization refactoring opportunities. Second, *CCFinder* is widely used and recognized. It has announced more than 5000 licenses.[10] It was awarded *Clone Awards* in 2002.

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. The number of involved experimenters (seven persons) is not large, which is also a threat to external validity.

A threat to internal validity is that the proposed and existing approaches were not applied by the same experimenters. In the evaluation, refactoring opportunities recommended by different approaches were checked manually by experimenters. If an experimenter checked a project by using clone-based approach first, and then checked it again by using the proposed approach, the comparison would be unfair because of *effect of testing*. Therefore, we asked different experimenters to check the same applications by using different approaches. However, the differences in experimenters might be a threat to internal validity because the evaluation results might be determined by the differences in experimenters instead of differences in approaches. To reduce the threat, we exchanged roles of the experimenters when evaluation subjects switched from *JFace* to *Thout Reader* (see Sect. 7.3).

A threat to the construct validity is that if the experimenters knew the expectation of the evaluation in advance, the experimenter using the proposed approach would work harder, whereas the other using existing approaches would slack off. It might also influence their decision on the recommended refactoring opportunities: The experimenter using the proposed approach would tend to refactor more to meet the expectation of the authors. To reduce the threat, we isolated them from each other, and they were not informed that they were participating a comparison.

A threat to external validity comes from evaluation subjects. Specific characteristics of the subjects may affect the evaluation results. To reduce the threat, we carried out the evaluation on 5 applications from different domains developed by different developers.

[10] http://www.ccfinder.net/index.html.

GenReferee has some parameters (thresholds) that might influence its precision and recall. In Sect. 7.2, we have calibrated them on two open-source applications, *GEF* (middle-sized) and *JSM* (small-sized). With the calibration, the precision of *GenReferee* is increased by nearly 50%, whereas its recall is not reduced. However, if further calibration could be done on more applications, *GenReferee* might be more accurate and efficient. Future work on this topic is briefly discussed in Sect. 9.

8 Discussion

While comparing the proposed approach to clone-based approach, we select *CCFinder* as the representative of clone detection tools. Though the evaluation results suggest that the proposed approach is more effective and efficient in detecting generalization refactoring opportunities than *CCFinder*-based approach, it is possible that the conclusion would not hold if other clone detection tools are employed. Consequently, further comparison against other clone-detection tools is desirable.

While extracting words from names of classes, methods, and parameters, the proposed approach assumes that the names follow the widely used camel-case or Pascal-case capitalization styles. The assumption makes the approach simple and easy to implement. However, the assumption does not always hold. If source code does not follow the styles, word segmentation technologies are needed.

While comparing attributes (methods, as well), type comparison ignores relationships between different types, i.e., inheritance. For example, *GenReferee* would report the type similarity $S_{type} = 0$ for attributes *Leader:Person* of class *TeachingTeam* and *Leader:Student* of class *ContestTeam*, though *Student* is a subtype of *Person* (presented on the left part of Fig. 6). The reasons are explained as follows. First, if the two attributes should be unified by pulling them into the super class *Team* (a generalization refactoring), there are two solutions as presented in Fig. 6. In *Solution B*, some methods or attributes of *Student* would never be used by instances of *TeachingTeam*, which throws doubts on benefits of the generalization refactoring. *Solution A* is even worse because syntax errors would be reported if instances of *ContestTeam* refer methods or attributes defined in *Student* but not in *Person*. As a conclusion, only if no instance of *ContestTeam* would access these methods or attributes, the generalization refactoring can and should be conducted. *GenReferee* can detect this refactoring opportunity though it does not analyze these complex type hierarchies. The reason is that software engineers can apply existing type constrain analysis (Tip et al. 2003) first, which would turn *Leader:Student* of class *ContestTeam* into *Leader:Person* because no unique methods or attributes of *Student* are accessed by instances of *ContestTeam*. After that, types of instances are minimized, and *GenReferee* does not have to analyze inheritances among types to detect common method and attributes.

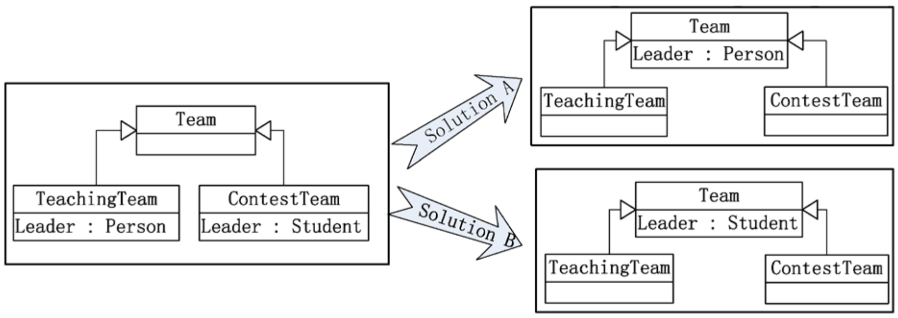


Fig. 6 Comparison of Types

Generalization refactoring would not necessitate the adjustment of clients of the refactored source code: no method, attribute, or interface would disappear. But it is possible to restructure clients to make full use of the new inheritance hierarchies. For example, if a new interface is introduced, developers might retype some variables with this new interface. As a result, the interface would decouple clients from their servers. Tip et al. (2003) have proposed an approach to identify which variables can be retyped.

Formula (1) gives equal weights to attribute similarity and method similarity. The reason is that if either of them is given a significantly greater weight, some potential refactoring opportunities might be missed. If the weight of attribute similarity is much smaller than that of method similarity, class pairs sharing common attributes only would not be identified. Similarly, class pairs sharing common methods only would be missed if the weight of method similarity is much smaller than that of attribute similarity. While discussing this issue with experienced engineers, we do not find any significant bias toward or against eight of them. Consequently, we give them the same weights. For the same reasons, signature similarity and implementation similarity are equally weighted in Formulas in Sect. 6.

9 Conclusion and future work

Generalization refactorings are widely used, and their benefits have been recognized (Fowler et al. 1999; Tip et al. 2003). However, detecting generalization refactoring opportunities is difficult, and we lack tool support for the task. To this end, we proposed an approach in this paper to semi-automatically identify generalization opportunities. The approach selects pairwise classes according to their conceptual relationship, implementation similarity, structural correspondence, and inheritance hierarchies. The approach has been implemented and evaluated on five non-trivial open-source applications. Evaluation results suggest that the approach could detect more generalization refactoring opportunities with less working effort than existing approaches. The main contribution of this paper is an approach to recommending generalization refactoring opportunities, an implementation of the approach, and an initial evaluation.

We plan to integrate the proposed tool *GenReferee* with *Breakaway* and *Eclipse* refactoring tool. Currently, *GenReferee* has been integrated into *Eclipse*, but it has not yet been integrated with *Breakaway* or *Eclipse* generalization refactoring tool. The integration would make generalization refactoring much easier: developers do not have to switch among the three tools.

We also plan to propose an approach to adjusting thresholds of *GenReferee* automatically. As discussed in Sect. 8, to make *GenReferee* more effective and efficient, we should do more calibration on more applications. However, the calibration as shown in Sect. 7.2 is time and resource-consuming. Since the final decisions made by developers could be tracked automatically, the thresholds might be adjusted automatically according to the decisions.

Currently, comments of methods and attributes are not considered by *GenReferee*. In the future, we should explore how to improve the approach by considering these comments.

The benefit (reduction in software cost and improvement in software quality) was not yet evaluated. According to existing research works (Kamiya et al. 2002; Tip et al. 2003), generalization refactoring could improve software extensibility and maintainability, and thus reduce software cost. But concrete benefit could be only measured with respect to future evolution scenarios.

Acknowledgements The authors from Beijing Institute of Technology are sponsored by the National Natural Science Foundation of China (No. 61003065), 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). The authors from Peking University are sponsored by the National Basic Research Program of China (No. 2011CB302604), and Beijing Municipal Natural Science Foundation (No. 4122036).

References

Adnan, R., Graaf, B., van Deursen, A., Zonneveld, J., Zonneveld, J.: Using cluster analysis to improve the design of component interfaces. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), pp. 383–386. IEEE Computer Society, Washington (2008). doi:10.1109/ASE.2008.54

Baker, B.S.: A program for identifying duplicated code. Comput. Sci. Stat. **24**, 49–57 (1992) citeseer.ist.psu.edu/baker92program.html

Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., Gueheneuc, Y.: Playing with refactoring: identifying extract class opportunities through game theory. In: IEEE International Conference on Software Maintenance (ICSM, 2010), pp. 1–5 (2010), doi:10.1109/ICSM.2010.5609739

Bavota, G., Lucia, A.D., Oliveto, R.: Identifying extract class refactoring opportunities using structural and semantic cohesion measures. J. Syst. Softw. **84**(3), 397–414 (2011). doi:10.1016/j.jss.2010.11.918

Cottrell, R., Chang, J.J.C., Walker, R.J., Denzinger, J.: Determining detailed structural correspondence for generalization task. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 165–174. ACM, New York (2007)

Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00), pp. 166–177. ACM, New York (2000). doi:10.1145/353171.353183

Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: Proceedings of 20th European Conference on Object-Oriented Programming (ECOOP'06), Lecture Notes in Computer Science, vol. 4067, pp. 404–428, Nantes, France. Springer, Berlin (2006)

Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining source code to automatically split identifiers for software analysis. In: 6th IEEE International Working Conference on Mining Software Repositories 2009 (MSR'09), pp. 71–80 (2009). doi:10.1109/MSR.2009.5069482

Feild, H., Binkley, D., Lawrie, D.: An empirical comparison of techniques for extracting concept abbreviations from identifiers. In: Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006) (2006), Dallas, Texas, USA

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)

Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE Trans. Softw. Eng. **31**, 166–181 (2005). doi:10.1109/TSE.2005.28

Guerrouj, L., Di Penta, M., Antoniol, G., Gueheneuc, Y.G.: Tidier: an identifier splitting approach using speech recognition techniques. J. Softw. Maint. Evol. (2011). doi:10.1002/smr.539

Higo, Y., Ueda, Y., Kamiya, T., Kusumoto, S., Inoue, K.: On software maintenance process improvement based on code clone analysis. In: Product Focused Software Process Improvement, Lecture Notes in Computer Science, vol. 2559. Springer, Berlin (2002), pp. 185–197. http://dx.doi.org/10.1007/3-540-36209-6_17

Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Aries: refactoring support environment based on code clone analysis. In: Proceedings of the Eighth IASTED Internal Conference Software Engineering and Applications, Cambridge, MA, USA, pp. 222–229 (2004)

Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Aries: refactoring support tool for code clone. Softw. Eng. Notes **30**, 1–4 (2005). doi:10.1145/1082983.1083306

Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: an approach to recommend relevant examples. IEEE Trans. Softw. Eng. **32**(12), 952–970 (2006)

Hunt, J., Tichy, W.: Extensible language-aware merging. In: Proceedings. International Conference on Software Maintenance, 2002, pp. 511–520 (2002). doi:10.1109/ICSM.2002.1167812

Hunt, J.W., McIlroy, M.D.: An algorithm for differential file comparison. Tech. rep. 41, Computing science technical report, Bell Laboratories, Murray Hill, New Jersey, USA (1976)

Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multi-linguistic token based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(6), 654–670 (2002)

Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in java. In: 30th International Conference on Software Engineering, Leipzig, Germany. ACM, New York (2008), pp. 431–440

Kim, M., Notkin, D.: Program element matching for multi-version program analyses. In: Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR'06), ACM, New York (2006), pp. 58–64. doi:10.1145/1137983.1137999

Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: IEEE 31st International Conference on Software Engineering, 2009 (ICSE 2009), pp. 309–319 (2009). doi:10.1109/ICSE.2009.5070531

Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), IEEE Computer Society, Washington (2007), pp. 333–343. doi:10.1109/ICSE.2007.20

Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: 13th Working Conference on Reverse Engineering, pp. 253–262 (2006)

Laski, J., Szermer, W.: Identification of program modifications and its applications in software maintenance. In: Proceerdings of Conference on Software Maintenance, pp. 282–290 (1992). doi:10.1109/ICSM.1992.242533

Madani, N., Guerrouj, L., Di Penta, M., Gueheneuc, Y., Antoniol, G.: Recognizing words from source code identifiers using speech recognition techniques. In: 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), pp. 68–77 (2010). doi:10.1109/CSMR.2010.31

Mens, T., Touwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)

Moha, N., Gueheneuc, Y.G., Leduc, P.: Automatic generation of detection algorithms for design defects. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp. 297–300 (2006). doi:10.1109/ASE.2006.22

Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F.: Decor: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2010). doi:10.1109/TSE.2009.50

Moore, I.: Guru- a tool for automatic restructuring of self inheritance hierarchies. In: International Conference on Technology of Object-Oriented Languages and Systems, pp. 267–275. Prentice-Hall, New York (1995)

Munro, M.: Product metrics for automatic identification of “bad smell” design problems in java source-code. In: 11th IEEE International Symposium Software Metrics, 2005, pp. 15–19 (2005). doi:10.1109/METRICS.2005.38

Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)

Revelle, M., Gethers, M., Poshyvanyk, D.: Using structural and textual information to capture feature coupling in object-oriented software. Empir. Softw. Eng. **16**, 773–811 (2011). doi:10.1007/s10664-011-9159-7

Sager, T., Bernstein, A., Pinzger, M., Kiefer, C.: Detecting similar java classes using tree algorithms. In: Proceedings of the 2006 International Workshop on Mining Software Repositories International, pp. 65–71 (2006)

Snelting, G., Tip, F.: Reengineering class hierarchies using concept analysis. In: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT ’98/FSE-6), pp. 99–110. ACM, New York (1998). doi:10.1145/288195.288273

Steimann, F.: The infer type refactoring and its use for interface-based programming. J. Object Technol. **6**(2), 99–120 (2007)

Streckenbach, M., Snelting, G.: Refactoring class hierarchies with kaba. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’04), pp. 315–330. ACM, New York (2004). doi:10.1145/1028976.1029003

Tip, F., Kiezun, A., Baeumer, D.: Refactoring for generalization using type constraints. In: Proceedings of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’03), Anaheim, CA, pp. 13–26 (2003)

Tourwe, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR’03), pp. 91–100 (2003)

Travassos, G., Shull, F., Fredericks, M., Basili, V.R.: Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’99), pp. 47–56. ACM, New York (1999). doi:10.1145/320384.320389

Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans. Softw. Eng. **99**, 347–367 (2009). doi:10.1109/TSE.2009.1

Tsantalis, N., Chatzigeorgiou, A.: Identification of refactoring opportunities introducing polymorphism. J. Syst. Softw. **83**(3), 391–404 (2010)

Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. J. Syst. Softw. **84**, 1757–1782 (2011). doi:10.1016/j.jss.2011.05.016

Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M.: On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans. Softw. Eng. **33**(12), 800–817 (2007) doi:10.1109/TSE.2007.70745

Wake, W.C.: Refactoring Workbook. Addison Wesley, Reading (2003)

Wang, Z., Pierce, K., McFarling, S.: Bmat—a binary matching tool for stale profile propagation. J. Instr.-Level Parallelism **2**, 1–20 (2000)

Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE’05), pp. 54–65. ACM, New York (2005). doi:10.1145/1101908.1101919

Yang, L., Liu, H., Niu, Z.: Identifying fragments to be extracted from long methods. In: Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference, pp. 43–49. IEEE Computer Society, Washington (2009). doi:10.1109/APSEC.2009.20

Yang, W.: Identifying syntactic differences between two programs. Softw. Pract. Exp. **21**, 739–755 (1991). doi:10.1002/spe.4380210706

Zhang, M., Hall, T., Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Maint. Evol. **23**(3), 179–202 (2011). doi:10.1002/smr.521

Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. IEEE Trans. Softw. Eng. **31**, 429–445 (2005). doi:10.1109/TSE.2005.72