Requirements Eng (2014) 19:309–329
DOI 10.1007/s00766-013-0197-0

RE 2013

Supporting requirements to code traceability through refactoring

Anas Mahmoud · Nan Niu

Received: 26 July 2013 / Accepted: 11 December 2013 / Published online: 23 December 2013

Abstract In this paper, we hypothesize that the distorted traceability tracks of a software system can be systematically re-established through refactoring, a set of behavior-preserving transformations for keeping the system quality under control during evolution. To test our hypothesis, we conduct an experimental analysis using three requirements-to-code datasets from various application domains. Our objective is to assess the impact of various refactoring methods on the performance of automated tracing tools based on information retrieval. Results show that renaming inconsistently named code identifiers, using Rename Identifier refactoring, often leads to improvements in traceability. In contrast, removing code clones, using eXtract Method (XM) refactoring, is found to be detrimental. In addition, results show that moving misplaced code fragments, using Move Method refactoring, has no significant impact on trace link retrieval. We further evaluate Rename Identifier refactoring by comparing its performance with other strategies often used to overcome the vocabulary mismatch problem in software artifacts. In addition, we propose and evaluate various techniques to mitigate the negative impact of XM refactoring. An effective traceability sign analysis is also conducted to quantify the effect of these refactoring methods on the vocabulary structure of software systems.

Keywords Information retrieval · Traceability · Refactoring

A. Mahmoud (✉) · N. Niu
Department of Computer Science and Engineering,
Mississippi State University, Mississippi State, MS 39762, USA
e-mail: amm560@msstate.edu

N. Niu
e-mail: niu@cse.msstate.edu

1 Introduction

Modern traceability tools employ information retrieval (IR) methods for automated support [20, 24, 48]. Such methods aim to match a query of keywords with a set of artifacts in the software repository and rank the retrieved artifacts based on how relevant they are to the query using a predefined similarity measure. The main assumption is that a coherent vocabulary structure, derived from the system's application domain, has been used throughout the system's life cycle [41, 43]. This vocabulary structure is embedded in the nonformal features of source code, or the attributes of the code that do not have an influence on the functionality of the system (e.g., variables and methods names, code comments, and messages) [2], and the textual content of software artifacts (e.g., requirements and design documents). Therefore, artifacts with similar vocabulary probably share several concepts, so they are likely candidates to be traced from one another [4]. However, as projects evolve, new and inconsistent terminology gradually finds its way into the system [58], causing topically related system artifacts to exhibit a large degree of variance in their contents [3, 33]. This phenomena is known as the *vocabulary mismatch* problem and is regarded as one of the principal causes of declining accuracy in retrieval engines [23].

A suggested solution for the vocabulary mismatch problem is to systematically recover the decaying vocabulary structure of the system through refactoring. Refactoring refers to a set of behavior-preserving transformations that improve the quality of a software system without changing its external behavior [78]. These transformations act on the internal structure of software artifacts, including the nonformal and organizational features in the system, leaving the system's functionality intact [36]. Refactoring

is now being advocated as an essential step in software development. For example, in agile methods, refactoring has already been integrated as a regular practice in the software's life cycle [71]. In addition, refactoring tools, which support a large variety of programming languages, have been integrated into most popular integrated development environments (IDEs), targeting various quality aspects of software systems (e.g., increase maintainability, reusability, and understandability) [14, 36, 51, 53, 63, 72, 73]. Motivated by these observations, in this paper, we hypothesize that certain refactoring methods will help to re-establish the system's vocabulary structure that often gets corrupted during evolution [58], thus improving the retrieval capabilities of IR methods operating on that structure.

Refactoring can take different forms affecting different types of artifacts. Therefore, testing our research hypothesis entails addressing several sub-research questions such as: what refactoring methods improve trace retrieval quality? What refactoring methods have more influence on the system's traceability? How to evaluate such influence? How does refactoring compare with other performance enhancement strategies in automated tracing? And how to reverse any potential negative impact certain refactoring methods might have on traceability? To answer these questions, we conduct an experimental analysis using three datasets from various application domains. Our main objective is to explore systematic ways for enhancing the performance of IR-based automated tracing tools.

In our previous work [63], we discovered that out of three refactoring methods studied, including: Rename Identifier, eXtract Method (XM), and Move Method (MM), only Rename Identifier led to improvements in traceability, while the XM refactoring, which targets code clones in the system, was found to be detrimental. In addition, moving misplaced code fragments, using MM refactoring, was found to have no significant impact on the performance. In this extension of our previous paper, we extend our analysis from solely measuring the effect of different refactoring methods on the tracing performance to more thoroughly investigating the internal operation of such methods. In particular, we first report the original experiment which we conducted to evaluate the impact of these three refactoring methods and then propose and evaluate mitigating strategies for minimizing the negative effect of method extraction on trace retrieval. Furthermore, based on our previous findings that renaming identifiers improves tracing results, we investigate whether it is better to perform this task as a refactoring activity (i.e., by modifying the code) or through a competing approach.

The rest of this paper is organized as follows. Section 2 presents a theoretical foundation of IR-based automated tracing. Section 3 introduces refactoring and describes the different refactoring methods used in our analysis. Section 4 describes our research methodology and experimental design. Section 5 presents and discusses the results. Section 6 describes the study limitations. Section 7 reviews related work. Finally, Sect. 8 concludes the paper and discusses directions for future work.

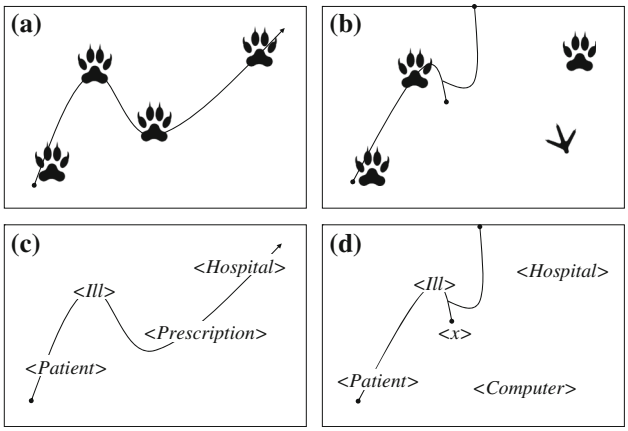

Fig. 1 Illustration of sign tracking. **a** A continuous track in the wild. **b** A distorted track in the wild. **c** A continuous track in the system. **d** A distorted track in the system

2 IR-based automated tracing

To understand the mechanism of IR-based automated tracing tools, we refer to the main theory underlying IR-based trace link retrieval. In their vision paper, Gotel and Morris [43] established an analogy between animal tracking in the wild and requirements tracing in software systems. This analogy is based on reformulating the concepts of *sign*, *track* and *trace*. A sign in the wild is a physical impact of some kind left by the animal in its surroundings, e.g., a footprint. Figure 1a shows a continuous track of footprints left by a certain mammal. The task of the hunter is to trace animals' tracks by following these signs. In other words, to trace means basically to follow a track made up of a continuous line of signs. Similarly, in requirements tracing, a sign could be a term related to a certain domain concept, left by a software developer or a system engineer in a certain artifact. Figure 1c shows a continuous track of related words from the health care domain $\langle Patient, Ill, Prescription, Hospital \rangle$. The task of IR methods is to trace these terms to establish tracks in system. These continuous tracks are known as *links*.

The availability of uniquely identifying marks, or signs, is vital for the success of the tracing process. However, just as in the wild, tracks in software systems can get discontinued or distorted due to several practices related to software evolution [32, 58]. In what follows, we identify three

symptoms related to code decay that might lead to such a problem. These symptoms include:

- *Missing signs* A track can get discontinued when a concept-related term in a certain artifact is lost. Figure 1d shows how the trace link becomes discontinued when the word ⟨*Prescription*⟩ is changed to ⟨*x*⟩. This can be equivalent to a footprint being washed off by rain in the wild (Fig. 1b).
- *Misplaced signs* A track can also be distorted by a misplaced sign. For example, the word ⟨*Computer*⟩, which supposedly belongs to another track, is positioned in the track of Fig. 1d. In the wild, this is equivalent to a footprint implanted by another animal on the track of unique footprints left by the animal being traced (e.g., Fig. 1b shows a bird's footprint left on the mammal's track in Fig. 1a).
- *Duplicated signs* This phenomenon is caused by the fact that some identical or similar code fragments are replicated across the code. These fragments are known as code clones 15. In our example, this can be equivalent to a track branching into some other module that contains a word similar to one of the signs of the trace link identified in Fig. 1c. Some animals adopt this strategy in the wild to confuse their predators by duplicating their footprints in different directions at different periods of time.

Our conjecture in this paper is that refactoring will help to reverse the effect of these symptoms, thus systematically re-establishing traceability tracks in the system.

3 Refactoring

Refactoring was initially introduced by Opdyke and Johnson [79] as a systematic means for aiding evolution and reuse in legacy software systems. While it can be applied to various types of artifacts, such as design and requirements, refactoring is mostly known for affecting source code [71]. Program refactoring starts by identifying *bad smells* in source code. Bad smells are "structures in the code that suggest the possibility of refactoring" [36]. Once refactoring has been applied, special metrics can be used to determine the effect of changes on the quality attributes of the system, such as maintainability and understandability [90].

A comprehensive catalog of code refactoring methods can be found at http://refactoring.com/. Refactoring can be manual, semi, or fully automated. Manual refactoring requires software engineers to synthesize and analyze code, identify inappropriate or undesirable features (code smells), suggest proper refactorings for these issues, and perform potentially complex transformations on a large number of entities manually. Due to the high effort associated with such a process, the manual approach is often described as repetitive, time-consuming, and error-prone [69]. The semi-automated approach is what most contemporary IDEs implement. Under this approach, refactoring activities are initiated by the developer. The automated support helps to carry out the refactoring process, such as locating entities for refactoring and reviewing refactored results. In contrast, the fully automated approach tries to initiate refactoring by automatically identifying bad smells in source code and carrying out necessary transformations automatically. However, even in fully automated tools, the final decision whether to accept or reject the outcome of the refactoring process is left to the human [51].

Deciding on which particular refactoring to apply to a certain code smell can be a challenge. In fact, applying arbitrary transformations to a program is more likely to corrupt the design rather than improving it [78]. However, there is no agreement on what transformations are most beneficial and when they are best applied. In general, such decisions should stem from the context of use, such as the characteristics of the problem, the cost-benefit analysis, or the goal of refactoring (e.g., improving robustness, extensibility, reusability, understandability, or performance of the system) [69, 71]. In automated tracing, the main goal of adopting refactoring is to improve the system's vocabulary structure in such a way that helps IR-based tracing methods to recover more accurate lists of candidate links. Based on that, we define the following requirements for integrating refactoring in the IR-based automated tracing process:

- *Altering nonformal information of the system* As mentioned earlier, IR-based tracing methods exploit nonformal information embedded in the textual content of software artifacts 2. Therefore, for any refactoring to have an impact on IR-based automated tracing methods, it should directly affect the system's textual content.
- *Coverage* Traceability links are often spread all over the system, linking a large number of the system's artifacts through various types of traceability relations [83]. Therefore, statistically speaking, to have a noticeable impact on the performance, adopted refactorings shall affect as many software entities as possible.
- *Automation* Since the main goal of automated tracing tools is to reduce the manual effort, any integrated refactoring should allow automation to a large extent. For any refactoring process to be considered effort-effective, it should provide automated solutions for code smell detection and applying code changes [35]. Automating these two steps will help to alleviate a large portion of effort usually associated with manual refactoring.

- *Granularity level* In all of our experimental datasets, traceability links are established at class granularity level (i.e., requirements-to-class) [48]. This limits our analysis in this paper to refactorings that work within the class scope (e.g., MM and XM), rather than refactorings that affect the class structure of the system (e.g., Remove Class or Extract Class). Enforcing this requirement ensures that our gold-standard remains unchanged after applying various refactorings.

Based on these requirements, we identify three categories of refactoring that might have an impact on the performance of IR-based tracing methods. These categories include refactorings that restore, remove, and move textual information in the system, represented by Rename Identifier, XM, and MM refactorings, respectively. These particular refactoring methods have been reported to be among the most understood and commonly used refactorings in practice [1, 29, 73, 75]. In addition, the research on the automation of these particular refactorings have noticeably excelled in the past few years, producing a wide selection of tools that support a large number of programming environments in a scalable manner [51, 74, 81, 91]. Therefore, we select these particular refactorings as a target of our investigation in this paper. In what follows we describe each of these refactorings in greater detail.

3.1 Restoring information

Refactoring methods under this category target the degrading vocabulary structure of source code [3, 58]. The main goal is to restore the domain knowledge that often gets lost over iterations of system evolution. In general, any refactoring that results in adding new words to the set of the system's vocabulary can be classified under this category. For example, refactorings such as Add Parameter or Introduce Explaining Variable introduce new variables or parameters, thus potentially new domain-related knowledge. However, the most popular refactoring in this category is Rename Identifier (RI) [1, 29, 73, 75]. As the name implies, this transformation refers to simply renaming an identifier (e.g., a variable, class, structure, method, or field) to give it a more relevant name [36]. Rename Identifier is expected to target the *Missing Sign* problem affecting traceability methods.

As mentioned earlier, to be considered in our analysis, refactoring methods should provide support for automatic detection of code smells they target. In our analysis, we refer to the literature of source code abbreviations and acronyms expansion to identify procedures for capturing opportunities for Rename Identifier refactoring [16, 56, 89]. In particular we apply the following procedure:

1. Identifiers are first divided into their constituent parts for analysis [56].
2. Identifiers with <4-character length. These are usually acronyms or abbreviations. In that case, the long form is used. For example, the parameter *HCP* in our health care system is expanded to *HealthCarePersonnel*. If the identifier is less than 4 characters but it is not an acronym nor an abbreviation, then it is renamed based on the context.
3. Identifiers which have a special word as part of their names. For example, the variable *PnString* is expanded to *PatientNameString*.
4. Identifiers with generic names. For example, in our health care system, the method's name *import* is expanded to *importPatientRecords*.

The main objective of this procedure is to achieve consistency. During multiple iterations of software evolution, slightly different abbreviations might be used to refer to the same domain concept, causing a mismatch between the vocabulary used in source code and that used in other software artifacts [57, 58]. This phenomenon is often described as a very occurring problem in software maintenance [27, 55, 56]. The proposed procedure for renaming identifiers tries to eliminate this inconsistency in the system by using one consistent form, whether an abbreviation or an extended form, to refer to the same domain concept. In our analysis, we use the extended full-word form. Our decision is based on the converging evidence from related literature which indicates that, in the long run, abbreviations impact comprehension negatively [44, 87]. In contrast, full-word identifiers often lead to the best comprehension [21, 57]. In addition, using the long form keeps identifiers' names in sync with their functionality, which results in an overall improvement in code quality, and thus the accuracy of IR methods working with these identifiers.

We implement our procedure to find candidates for renaming in our datasets. Once the candidate identifiers for renaming have been identified, the refactoring tool available in eclipse 4.2.1 IDE is used to carry out the renaming process. This will ensure that all corresponding references in the code are updated automatically. Finally, the code is compiled to make sure no bugs were introduced during the process. It is important to point out that at the current stage of the research, choosing new identifiers' names is still a manual task, carried out by our researchers, using keywords available in the system's documentation, based on their understanding of the system's application domain, and the particular functionality of the identifier being renamed.

3.2 Moving information

This category of refactoring methods is concerned with moving code entities between system modules. The goal is to reduce coupling and increase cohesion in the system,

which is a desired quality attribute of object-oriented design [39]. Refactorings under this category provide a remedy against the *Feature Envy* code smell. An entity has *Feature Envy* when it uses, or being used by, the features of a class other than its own (different from where it is declared). This may indicate that the entity is misplaced [36].

In our experiment, we adopt MM refactoring as a representative of this category. By moving entities to their correct place, this particular refactoring is expected to target the *Misplaced Sign* problem mentioned earlier. To identify potentially misplaced entities, we adopt the strategy proposed by Tsantalis and Chatzigeorgiou [91], in which they introduced a novel entity placement metric to quantify how well entities have been placed in code. This semi-automatic strategy starts by identifying the set of the entities each method accesses (parameters or other methods). *Feature Envy* code smell instances are then detected by measuring the strength of coupling that a method has to methods belonging to all foreign classes. The method is then moved to the target foreign class in such a way that ensures that the behavior of the code will be preserved. This procedure has been implemented as an ECLIPSE plug-in (*Jdeodorant*[1]) that identifies *Feature Envy* instances and allows the user to apply the refactorings that resolve them. However, despite of the high degree of automation, this process can still be regarded as semi-automatic [91]. In particular, verifying or rejecting the MM candidates suggested by the tool, and making sure that moving a method does not introduce any bugs in the system, are still manual tasks. In our analysis, we only consider misplaced methods. *Move Attribute* refactoring is excluded based on the assumption that attributes have stronger conceptual binding to the classes in which they are initially defined; thus, they are less likely than methods to be misplaced [91].

3.3 Removing information

These refactorings remove redundant or unnecessary code in the system. A popular code smell such refactorings often handle is *Duplicated Code*. This code smell is usually produced by Copy-and-Paste programming [52], and indicates that the same code structure appears in more than one place. These duplicated structures are known as code clones and are regarded as one of the main factors for complicating code maintenance tasks [68]. Exact duplicated code structures can be detected by comparing text [36]. However, other duplicates, where entities have been renamed or the code is only functionally identical, need more sophisticated techniques that work on the code semantics rather than its lexical structure [30].

The most frequent way to handle code duplicates is XM refactoring [67, 92]. In particular, for each of the duplicated blocks of code, a method is created for that code, and then all the duplicates are replaced with calls to the newly extracted method. When the duplicates are scattered in multiple classes, the new extracted method is assigned to the class that calls it the most. By removing potentially ambiguous duplicates, XM is expected to target the *Duplicated Sign* problem of software artifacts. In our analysis, we use the duplicated code detection (SDD)[2] ECLIPSE plug-in to detect code clones. XM refactoring available in the ECLIPSE 4.2.1 IDE is then used to refactor candidate clones. In particular, the user selects the code fragment to be extracted from the list of candidate clones returned by the tool, and ECLIPSE will ask for a method name and a class to host the newly extracted method. A method name is selected based on the context of the code. Once the method is created, the user is responsible for replacing all clone instances with a call to the new method and making sure that no bugs are introduced by this process.

Table 1 summarizes the categories of refactoring introduced in this section, including the code smells they target, traceability problems they might impact, the tool support available, and a summary of the manual effort required to carry out each refactoring.

4 Methodology and research hypothesis

This section describes our research approach, including our experimental procedure, datasets used in conducting our experiment, and evaluation mechanisms to assess the performance.

4.1 Datasets

Three datasets were used to conduct the experiment in this paper including: *iTrust*, *eTour*, and *WDS*. Following is a description of these datasets and their application domains:

- *iTrust* An open source medical application, developed by software engineering students at North Carolina State University (USA). It provides patients with a means to keep up with their medical history and records and to communicate with their doctors [70]. The dataset (source code: v15.0, Requirements: v21) contains 314 requirements-to-code links. The links are available at method level. To conduct our analysis, the links granularity is abstracted to class level based on a careful analysis of the system.

[1] http://www.jdeodorant.org/.

[2] http://wiki.eclipse.org/Duplicated_code_detection_tool_(SDD).

Table 1 Refactorings used in our analysis

Refactoring	Code smell	Tracing problem	Tool support	Manual effort
Rename Identifier	Decaying vocabulary	Missing signs	Eclipse	Verifying candidates for renaming Selecting identifiers' names
Move Method	Feature envy	Misplaced sign	Jdeodorant Eclipse	Verifying move-method candidates Verifying the results
eXtract Method	Code clones	Duplicated signs	SDD Eclipse	Verifying code clone candidates Selecting extracted method's name Selecting host class Verifying the results

Table 2 Experimental datasets

Dataset	Domain	Contributor	LOC (K)	COM (K)	No. req.	No. SC	Links
iTrust	Health care	North Carolina State University	20.7	9.6	50	299	314
eTour	Tourism	University of Salerno	17.5	7.5	58	116	394
WDS	Job search	Industrial partner	44.6	10.7	26	521	229

- *eTour* An open source electronic tourist guide application developed by final year Master's students at the University of Salerno (Italy). The dataset contains 394 requirements-to-code links that were provided with the dataset. *eTour* was selected as an experimental system because its source code contains a combination of English and Italian words, which is considered an extreme case of vocabulary mismatch.
- *WDS* A proprietary software-intensive platform that provides technological solutions for service delivery and workforce development in a specific region of the United States. In order to honor confidentiality agreements, we use the pseudonym WDS to refer to the system. WDS has been deployed for almost a decade. The system is developed in Java and the current version has 521 source code files. For our experiment, we devise a dataset of 229 requirements-to-code links, linking a subset of 26 requirements to their implementation classes. These links were provided by the system's developers.

Table 2 shows the characteristics of each dataset. The table shows the size of the system in terms of lines of source code (LOC), lines of comments (COM), source and target of traceability links, i.e., number of requirements (No. Req.), number of code elements they link to (No. SC), and the number of correct traceability links.

4.2 Experimental procedure

Our experimental procedure can be described as a multi-step process as follows:

- *Refactoring* Initially the system is refactored using various refactoring methods mentioned earlier. The goal is to improve the system lexical structure before tracing. The results of applying different refactorings over our three experimental datasets are shown in Table 3. The table shows the number of entities affected by refactoring in each dataset (e.g., number of moved or extracted methods and number of renamed identifiers), the number of affected classes in each system, and the number of affected classes in the gold-standard, or classes that are part of a trace link in our answer sets (C').
- *Indexing* This process starts by extracting textual content (e.g., comments, code identifiers, requirements text) from input artifacts. Lexical processing is then applied (e.g., splitting code identifiers into their constituent words). Stemming is performed to reduce words to their roots. In our analysis we use Porter stemming algorithm [80]. The output of the process is a compact content descriptor, or a *profile*, which is usually represented as keywords components matrix or a vector space model (VSM) [62].
- *Retrieval* IR methods are used to identify a set of traceability links by matching the traceability query's profile with the artifacts' profiles in the software repository. Links with similarity scores above a certain threshold (cutoff) value are called candidate links [48]. In our experiment, we use Vector Space Model with TFIDF weights as our experimental baseline. VSM–TFIDF is a popular scheme in VSM which has been validated through numerous traceability studies as an experimental baseline (e.g., [48, 64]). Mathematically,

Table 3 Entities and classes affected by refactoring

Refactoring	iTrust			eTour			WDS		
	Entities	Classes	C'	Entities	Classes	C'	Entities	Classes	C'
Rename Identifier	175	113	110	85	63	57	203	174	166
Move Method	22	44	44	17	31	29	24	62	61
eXtract Method	132	201	193	45	92	88	62	102	98

when using VSM, each document is represented as a set of terms $T = \{t_1, \ldots, t_n\}$. Each term t_i in the set T is assigned a weight w_i. The terms in T are regarded as the coordinate axes in an N dimensional coordinate system, and the term weights $W = \{w_1, \ldots, w_n\}$ are the corresponding values. Thus, if q and d are two artifacts represented in the vector space, then their similarity is measured as the cosine of the angle between them as shown in (Eq. 1):

$$s(q,d) = \frac{\sum q_i \cdot d_i}{\sqrt{\sum q_i^2 \cdot \sum d_i^2}} \quad (1)$$

where q_i and d_i are real numbers standing for the TFIDF weight of term i in q and d, respectively. $q_i = tf_i(q) \cdot idf_i$ and $d_i = tf_i(d) \cdot idf_i$, where $tf_i(q)$ and $tf_i(d)$ are the frequencies of term i in q and d, respectively. idf_i is the inverse document frequency and is computed as $idf_i = log_2(t/df_i)$, where t is the total number of artifacts in the corpus, and df_i is the number of artifacts in which term i occurs.

- *Evaluation* At this step, different evaluation measures are used to assess the different aspects of the performance. In what follows, we describe these measures in greater detail.

4.3 Evaluation

Sundaram et al. [86] identified a number of primary and secondary measures to assess the performance of different tracing tools and techniques. These measures can be categorized into two groups as follows.

4.3.1 Quality measures

Precision (P) and Recall (R) are the standard IR measures to assess the quality of the different traceability tools and techniques. Recall measures coverage and is defined as the percentage of correct links that are retrieved, and precision measures accuracy and is defined as the percentage of retrieved links that are correct [66]. Formally, if A is the set of correct links and B is the set of retrieved candidate links, then Recall and Precision can be defined as:

$$R(\text{Recall}) = |A \cap B|/|A| \quad (2)$$

$$P(\text{Precision}) = |A \cap B|/|B|. \quad (3)$$

4.3.2 Browsability measures

Browsability is the extent to which a presentation eases the effort for the analyst to navigate the candidate traceability links. For a tracing tool or a method that uses a ranked list to present the results, it is important to not only retrieve the correct links but also to present them properly. Being set-based measures, precision and recall do not sufficiently capture information about the list browsability. To reflect such information, other measures are usually used. Assuming h and d belong to sets of system artifacts $H = \{h_1, \ldots, h_n\}$ and $D = \{d_1, \ldots, d_m\}$. Let C be the set of true links connecting d and h, $L = \{(d,h)|\text{sim}(d,h)\}$ is a set of candidate traceability links between d and h generated by the IR-based tracing tool, where sim(d, h) is the similarity score between d and h. L_T is the subset of true positives (correct links) in L, a link in this subset is described as (d, h). L_F is the subset of false positives in L, a link in this subset is described using the notion (d', h'). Based on these definitions, secondary measures can be described as:

- *Mean average precision* (*MAP*) is a measure of quality across recall levels [6]. For each query, a cutoff point is taken after each true link in the ranked list of candidate links. The precision is then calculated. Correct links that were not retrieved (false negatives) are given a precision of 0. The precision values for each query are then averaged over all the relevant links (true positives) in the answer set of that query ($|C|$), producing average precision (AP). The mean average precision (MAP) is calculated as the average of AP for all queries in each dataset [88]. MAP gives an indication of the order in which the returned documents are presented. For instance, if two IR methods retrieved the same number of correct links (same recall), then the method that places more true links toward the top of the list will have a higher MAP. Equation 4 describes MAP, assuming the dataset has Q traceability queries.

$$\text{MAP} = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{|C_j|}\sum_{k=1}^{|L_{T_j}|}\text{Precision}(L_{T_{j_k}}) \quad (4)$$

- *DiffAR* is a measure of the contrast of the list [85]. It can be described as the difference between the average similarity of true positives and false positives in a ranked list. A list with higher DiffAR has a clearer distinction between its correct and incorrect links hence is considered superior. Equation 5 defines DiffAR.

$$\text{DiffAR} = \frac{\sum_{(h,d)}\text{sim}(\text{h},\text{d})}{|L_T|} - \frac{\sum_{(h',d')}\text{sim}(\text{h}',\text{d}')}{|L_F|}. \quad (5)$$

Performance of each dataset after applying a certain refactoring, in comparison with the baseline (VSM), is presented as a precision/recall curve over various threshold levels ($\langle .1, .2, \ldots, 1\rangle$) [48]. A higher threshold level means a larger list of candidate links, i.e., more links, are considered in the analysis. Wilcoxon signed ranks test is used to measure the statistical significance of the results. This is a nonparametric test that makes no assumptions about the distribution of the data [28]. This test is applied over the combined samples from two related samples or repeated measurements on a single sample (before and after effect). The IBM SPSS Statistics software package is used to conduct the analysis. We use $\alpha = .05$ to test the significance of the results. Note that different refactorings are applied independently, so there is no interaction effect between them.

5 Results and discussion

This section starts by describing our analysis results. In particular, the effect of different refactoring methods on the performance in terms of preliminary measures (precision and recall) and browsability measures (MAP and DiffAR) is described. The section then proceeds by further exploring the effect of each refactoring method in greater detail. In particular, we compare the performance of methods that have positive impact on traceability with other related techniques in automated tracing, and explore strategies for mitigating any potential negative impact certain refactorings methods might have on the performance.

5.1 Analysis results

Figure 7 shows the recall and precision curves of our three datasets after applying RI, MM, and XM, in comparison with the VSM baseline. Analysis of variance over the results is shown in Table 4. In general, the results show that different refactorings vary in their impact on the performance. In details, RI refactoring has the most obvious positive impact on the results, affecting the recall significantly in all three datasets. In the *iTrust* dataset, both precision and recall have improved significantly, achieving optimal recall levels at higher thresholds. The same performance is detected in the *eTour* dataset, in which the improvement in the recall and the precision over the baseline is statistically significant. In the *WDS* dataset, the precision has dropped significantly with the significant increase in the recall. This can be explained based on the inherent trade-off between precision and recall. In this particular dataset, even though renaming identifiers has helped to retrieve more true positives, it also retrieved a high number of false positives.

The results also show that MM refactoring has the least influence on the performance. In all datasets no significant improvement in the recall or the precision is detected. In fact, the performance after applying this particular refactoring is almost equivalent to the baseline. In contrast, statistical analysis shows that XM has resulted in a significant increase in the precision. However, when applied, it was no longer possible to achieve high recall; hence, the performance lines in Fig. 7 stopped at recall of 66, 61, and 93 % in *iTrust*, *eTour*, and *WDS*, respectively. In general, In terms of recall, the results show that removing redundant textual knowledge from the system has caused a significant drop in the number of true links, taking the recall down to significantly lower levels in all three datasets. The spike in the precision can be simply explained based on the inherent trade-off between precision and recall.

In terms of browsability, statistical analysis in Table 4 shows that both RI and MM have no significant impact on the average DiffAR. However, XM seems to be achieving significantly better performance over the baseline. In terms of MAP, Fig. 2 shows the superior performance of XM over other refactorings in comparison with the baseline. This behavior can be explained based on the fact that VSM retrieves the smallest number of links after applying XM. However, even with lower recall, only a few false positives were separating true positives, with most of these true positives located toward the top the of list, thus taking the precision of these links to higher levels, which in turn resulted in higher MAP values (Eq. 4).

MAP results also show the inconstant performance of RI across the different datasets. In the *iTrust*, no significant difference in the performance is detected. In contrast, in the *eTour* dataset, RI achieves significantly better performance than the baseline and significantly worse performance in *WDS*. In addition, analysis results show that MM does not have any significant impact on the MAP values, which is actually expected based on the fact that it does not have a significant impact on the primary performance measures.

Table 4 Wilcoxon signed ranks test results ($\alpha = .05$) for primary performance measures

	iTrust		eTour		WDS	
	Recall (Z, p value)	Precision (Z, p value)	Recall (Z, $p$value)	Precision (Z, p value)	Recall (Z, $p$value)	Precision (Z, p value)
Refactorings						
Rename Identifier	(−2.395, <.010)	(−2.803, <.005)	(−2.701, <.007)	(−2.803, <.005)	(−2.090, <.05)	(−2.803, <.005)
Move Method	(−.405, .686)	(−1.599, .110)	(−1.753, .080)	(−.663, .508)	(−1.572, .116)	(.000, 1.000)
eXtract Method	(−2.803, <.005)	(−2.803, <.005)	(−2.701, <.007)	(−2.803, <.005)	(−2.803, <.005)	(2.701, <.007)
	MAP (Z, p value)	DiffAR (Z, p value)	MAP (Z, p value)	DiffAR (Z, p value)	MAP (Z, p value)	DiffAR (Z, p value)
Refactorings						
Rename Identifier	(−.357, .721)	(−1.732, .083)	(2.380, <.010)	(−1.414, .157)	(−2.803, <.005)	(−1.000, .317)
Move Method	(−.653, .514)	(.000, 1.000)	(1.478, .139)	(.000, 1.000)	(−1.680 .093)	(.000, 1.000)
eXtract Method	(−2.803, <.005)	(−2.842, <.005)	(−2.809, <.005)	(−2.803, <.005)	(−2.803, <.005)	(−3.051, <.005)

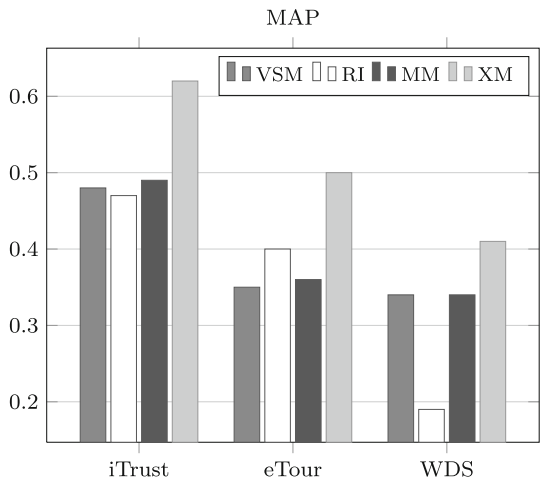

Fig. 2 MAP values in iTrust, eTour and WDS after applying different refactorings (*RI* Rename Identifier, *MM* Move Method, *XM* eXtract Method)

In general, our results suggest that RI refactoring has the most significant positive effect on the results, improving the recall to significantly higher levels in all three datasets. In contrast, XM has a significantly negative impact, taking the recall down to significantly lower levels in all three datasets, and MM has no clear impact on the performance. Automated tracing methods emphasize recall over precision [48]. This argument is based on the observation that an error of commission (false positive) is easier to deal with than an error of omission (false negative). Based on that, we conclude that RI refactoring has the most potential as a performance enhancement technique for IR-based requirements-to-code automated tracing. In what follows, the operation of each refactoring is discussed in greater detail.

5.2 Rename identifier effect

Our results suggest that restoring textual information has the most positive impact on the system's traceability. In particular, RI refactoring targets the vocabulary mismatch problem in software artifacts, which seems to be the most dominant problem affecting IR-based traceability tools [37]. In the automated tracing literature, the vocabulary mismatch problem is often handled by using semantics. In particular, techniques such as query expansion and thesaurus support are often used to fill the textual gap caused by poor coding habits [40, 48, 64]. Documents are then matched based on their lexical attributes as well as other semantic relations provided by such performance enhancement techniques.

In order to gain better insights into the operation of RI, in what follows, we compare the performance of these related performance enhancement strategies with the performance of VSM after applying RI refactoring on our datasets.

5.2.1 *Query expansion*

Query expansion is a technique in which the trace query is enriched with terms extracted from external knowledge sources in order to bridge the textual gap in the system. To implement query expansion, we use explicit semantic analysis (ESA) [38, 60]. ESA is a semantic relatedness technique that utilizes the textual content of Wikipedia to quantify the degree to which two concepts semantically relate to each other [38]. ESA represents the meaning of a text in a high dimensional weighted vector of concepts derived from Wikipedia, thus exploiting all different types of semantic links hidden in the text. Formally, given a text fragment $T = \langle t_1, \ldots, t_n \rangle$, and a space of Wikipedia articles

C, initially, a weighted vector V is created for the text, where each entry of the vector v_i is the *TFIDF* weight of the term t_i in T. Using a centroid-based classifier [46], all Wikipedia articles in C are ranked according to their relevance to the text. Let k_j be the strength of association of term t_i with Wikipedia article $c_j : c_j \in \{c_1, c_2, \ldots, c_n\}$ (where N is the total number of Wikipedia articles). The semantic interpretation vector S for text T can be described as a vector of length N, in which the weight of each concept c_j is defined as:

$$S_i = \sum_{w_{i \in T}} v_i \cdot k_j. \quad (6)$$

Entries of this vector reflect the relevance of the corresponding articles to text T. The relatedness between two texts can be calculated as the cosine between their corresponding vectors. ESA, as well as similar query expansion techniques, have been used before in automated tracing and have shown promising results in handling the vocabulary mismatch problem [40, 64].

5.2.2 Thesaurus support

Such techniques provide a more focused semantic support by utilizing the *synonymy* relations between different terms in the system's corpus. To add a thesaurus support to VSM, a domain-specific thesaurus is manually created. Such a thesaurus contains lists of synonym pairs derived from the project's application domain, and provides support for acronyms and abbreviations.

To integrate thesaurus support into our VSM baseline, for each pair of synonyms identified (s_i, s_j), a perceived similarity coefficient α_{ij} can be assigned to indicate their degree of equivalence. For each document in the corpus, document vectors are expanded based on these synonym pairs. A similarity coefficient of $\alpha_{ij} < 1$ is usually assigned to distinguish a *synonymy* match from an exact match where $\alpha_{ij} = 1$. Similarity $s(q, d)$ between two documents can then be calculated as [48]:

$$s(q,d) = \frac{\sum q_i \cdot d_i + \sum_{(k_i,k_j,\alpha_{ij}) \in T} \alpha_{ij}(q_i \cdot d_j + d_j \cdot q_i)}{\sqrt{\sum q_i^2 \cdot \sum d_i^2}}. \quad (7)$$

To approximate an acceptable value of α in Eq. 7., we propose an optimization algorithm that is based on maximizing the recall. The algorithm starts from $\alpha = 0$, gradually increasing this values by .05 each time, and monitoring the recall, looking for α values that achieve the highest average recall. Results show that an average similarity coefficient of $\alpha \approx .80$ achieves acceptable results in all three datasets.

To compare the performance of these two techniques with RI, we trace our datasets using both VSM with thesaurus support (VSM-T) and ESA, before applying RI refactoring, and compare their performance with the VSM baseline after applying RI (VSM–RI). The results are shown in Fig. 8 and Table 5. Results show that query expansion technique (ESA) was able to hit almost a 100 % recall at higher threshold levels in all datasets; however, the precision was affected negatively due to the high number of false positives. In general, textual enrichment of artifacts might have a positive influence on the recall, especially retrieving some of the hard-to-trace requirements [40]; however, it has a significant negative impact on the accuracy, which was reflected in the fast drop in the precision values at higher threshold levels. In contrast, the results show that VSM with a domain-specific thesaurus support was able to achieve a comparable performance to our refactoring-based approach; no statistically significant difference in terms of precision and recall was detected in all of our three datasets.

To demonstrate the operation of these three different techniques, we refer to the example in Fig. 3. This figure shows a true trace link between requirement 6.2.3, which describes a basic forgotten password recovery functionality, and the method FP_OnClick, which implements this particular requirement. Figure 4a shows the refactored method after applying our RI procedure described in Sect. 3.1. Figure 4b shows a snapshot of our domain-specific thesaurus. Basically, entries were added to handle abbreviations and basic domain-specific *synonymy* relations. Figure 4c shows the query expansion terms that have been added after applying ESA to both ends of the trace link. In particular, the link has been expanded with several unrelated terms, extracted from the general purpose knowledge source. For example, the list of semantically related terms for the term ⟨*forget*⟩ from requirement 6.2.6 includes many domain irrelevant terms such as ⟨*bury*, *leave*⟩. This explains the high noise-to-signal ratio returned by this method, causing the retrieval of a large number of incorrect links.

5.3 Handling code clones

A surprising observation in our analysis is that removing redundant information from software artifacts has a negative impact on the performance of IR-based automated tracing tools. This suggests that code clones actually serve a positive purpose for traceability link recovery. However, there is a conventional wisdom that code cloning is generally a bad development practice. From a refactoring perspective, code clones are considered a code smell 8. They significantly increase the maintenance cost and the error proneness of the code as inconsistent changes to code duplicates can lead to unexpected behavior. Therefore,

Table 5 Wilcoxon signed ranks test results ($\alpha = .05$) for query expansion and sign-preserving techniques

	iTrust		eTour		WDS	
	Recall (Z, p value)	Precision (Z, p value)	Recall (Z, p value)	Precision (Z, p value)	Recall (Z, p value)	Precision (Z, p value)
Technique						
VSM-T	(−.051, .959)	(−1.599, .110)	(−.652, .515)	(−.178, .859)	(−1.478, .139)	(−.866, .386)
ESA	(−2.490, <.05)	(−2.380, <.05)	(−2.803, <.005)	(−2.701, <.05)	(−2.842, <.005)	(−3.051, <.005)
Technique						
Summarization	(−.357, <.005)	(−2.842, <.005)	(−3.051, <.005)	(−2.803, <.005)	(−2.803, <.005)	(−2.090, <.01)
Labeling	(0, 1.0)	(0, 1.0)	(0, 1.0)	(0, 1.0)	(0, 1.0)	(0, 1.0)

(a)

6.2.3 Forget Password

The system should provide a functionality for recovering user's password. New password should be sent to the user's registered email.

(b)

```
1.  public bool FP_OnClick(string uName)
2.    {
3.        If(validteUsrNm(uName));
4.          stEmail = getUsrEml(uName);
5.          stPwd = genRndPWD();
6.          sendPwd(stEmail, stPwd);
7.          return true;
8.        else Error ="Invalid Credentials";
9.          return false;
10.   }
```

Fig. 3 **a** Trace link between requirement 6.2.3. **b** Method FP_OnClick

code clones have to be refactored whenever detected [54, 81].

5.3.1 Code summarization

To mitigate the impact of removing code clones on the system traceability, we suggest a sign-preserving treatment to reverse the negative effect of XM refactoring. Applying this treatment, whenever a redundant code (a code clone) is removed, appropriate comments that describe that code can be automatically inserted to fill the textual gap left by refactoring that particular code. This can be achieved by utilizing automatic techniques to generate descriptions for source code [49]. Several techniques for code labeling and summarization have been introduced in the literature [22, 45, 49]. Next we experiment with two commonly used techniques, at different levels of complexity, for preserving traceability signs.

Refers to the creation of a shortened version of a computer program by capturing and preserving the subject matter of the code [49]. Summarization is often performed

(a)

```
1.  public bool ForgetPassword_OnClick(string userName)
2.    {
3.        If(validteUserName(userName));
4.          stEmail = getUserEmail(userName);
5.          stPassword = generateRandomPassword();
6.          sendPassword(stEmail, stPassword);
7.          return true;
8.        else Error ="Invalid Credentials";
9.          return false;
10.   }
```

(b)

- ***pwd***: password
- ***credential***: email, password
- ***usr***: user

(c)

- ***credential***: *certificate, password, email*
- ***validate***: *formalize, formalize, corroborate*
- ***password***: *watchword, word, parole, countersign*
- ***recover***: *retrieve, find, regain, recuperate, reclaim*
- ***forget***: *bury, block, blank out, leave*

Fig. 4 Example of semantic support to overcome the vocabulary mismatch in the trace link in Fig. 3. **a** Applying Rename Identifier on Method FP_OnClick. **b** Domain thesaurus support. **c** ESA query expansion

with the objective of producing meaningful summaries of source code to aid program comprehension [45]. To generate code summaries of code clones we use latent semantic indexing (LSI) [26], a technique that is used very often for automatically extracting summaries of natural language. In particular, we adopt the approach proposed by Haiduc et al. [45] to extract the most important terms in a certain code. This approach has been shown to achieve high correlation with human-generated summaries [45]. The process starts by indexing the source code corpus at the method level. The cosine similarity is then computed between the code clone's profile and each of the terms in the corpus in the LSI-reduced space. The corpus terms are

(a)

```
1.    try {
2.        conn = factory.getConnection();
3.        ps = conn.prepareStatement("SELECT * FROM NDCodes, OVMedication, OfficeVisits "
4.            + "WHERE NDCodes.Code=OVMedication.NDCode AND OVMedication.VisitID=OfficeVisits.ID "
5.            + "AND PatientID=? AND ((DATE(?) < OVMedication.EndDate AND DATE(?) > OVMedication.StartDate)"
6.            + "OR (DATE(?) > OVMedication.StartDate AND DATE(?) < OVMedication.EndDate ) OR "
7.            + "(DATE(?) <= OVMedication.StartDate AND DATE(?) >= OVMedication.StartDate)) "
8.            + "ORDER BY VisitDate DESC");
9.        ps.setLong(1, patientID);
10.       ps.setString(2, startDate);
11.       ps.setString(3, startDate);
12.       ps.setString(4, endDate);
13.       ps.setString(5, endDate);
14.       ps.setString(6, startDate);
15.       ps.setString(7, endDate);
16.       ResultSet rs = ps.executeQuery();
17.       return loader.loadList(rs);
18.   } catch (SQLException e) {
19.       e.printStackTrace();
20.       throw new DBException(e);
21.   }
```

(b)

```
1. /* factory ND Code ID End */
2. byDate(patientID, startDate, endDate);
```

(c)

```
1. /** factory
2. * ND Code
3. * OV Medication
4. * Office Visit
5. * Visit ID
6. * Office Visit
7. * Patient ID
8. * End Date
9. * Star tDate
10. **/
11. byDate(patientID, startDate, endDate);
```

Fig. 5 Applying traceability sign preservation on a code clone. **a** A code clone detected in the *iTrust* dataset. **b** Comments generated by LSI-based code summarization technique. **c** Comments generated by the indexing-based code summarization technique

then ranked in decreasing order based on their similarity with the code clone. The summary is then constructed by considering the top N terms in the ranked list.

5.3.2 Code labeling

Refers to the extraction of a set of representative words for a certain code element. A simple code labeling can be achieved by indexing the redundant code (removing stop-words, splitting code identifiers into their constituent words, and performing stemming [22]). The resulting words are then added as comments to replace the removed code. The term *labeling* stems from the fact that no human-like meaningful descriptions are generated; instead, just discrete words (labels) are extracted to facilitate IR. Figure 5b, c show the outcome of applying the code summarization procedure and the code labeling procedure, respectively over a code clone detected in the *iTrust* dataset.

To evaluate the effectiveness of these two techniques in preserving traceability signs, we perform missing traceability sign analysis. In particular, we calculate the percentage of lost effective signs when removing code clones. We define an effective sign as a term or a keyword that contributes to a trace link (appears in both ends of the traceability link). Figure 6 shows that the indexing-based code labeling was actually more successful than code summarization in preserving a large number of the original signs lost after removing code clones. The best performance of the code summarization technique was achieved at ($N = 7$). The poor performance of code summarization can be explained based on the fact that size of a redundant code is not sufficient to produce meaningful summaries. For example, often the redundant code is just a part of a

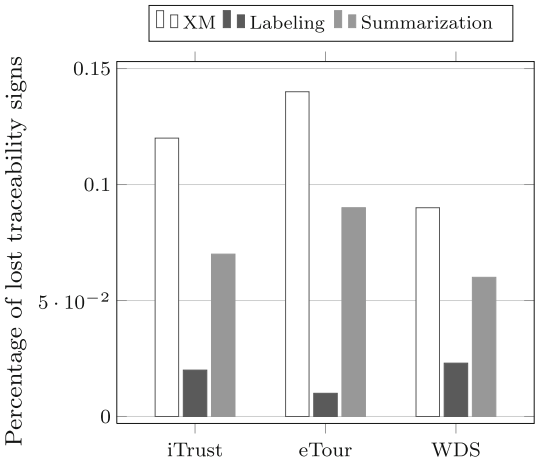

Fig. 6 Percentage of lost traceability signs in iTrust, eTour and WDS after applying eXtract Method (*XM*), and after applying code labeling and code summarization techniques ($N = 7$)

method. Such code fragments usually lack significant information that can be useful to the summarization process. For instance, in our example in Fig. 5a, the code clone does not include the method's signatures, which has been found to add a significant information value to the generated summaries [22, 45, 84]. These findings come actually aligned with previous observations that simple techniques were shown to better reflect the subject matter of the code than other more complicated techniques such as LSI or LDA [22] (Figs. 7, 8).

To further compare the effectiveness of these techniques, we integrate them into our experimental procedure after applying XM. We then re-trace all of our experimental datasets. Results are shown in Fig. 9 and Table 5. The results show that, when the indexing-based code labeling procedure is used, no statistically significant drop in terms of precision or recall is detected in any of our three datasets, i.e., the performance is unaffected by removing the clones. In contrast, while applying the LSI-based code summarization technique helps to preserve some of the effective traceability signs, it still could not improve the results significantly.

5.4 Moving information

Our results show that moving misplaced information among the system's modules has no significant impact on the performance. This suggests that misplaced signs might not be as problematic for IR-based traceability as missing or duplicated signs. This phenomenon can be explained based

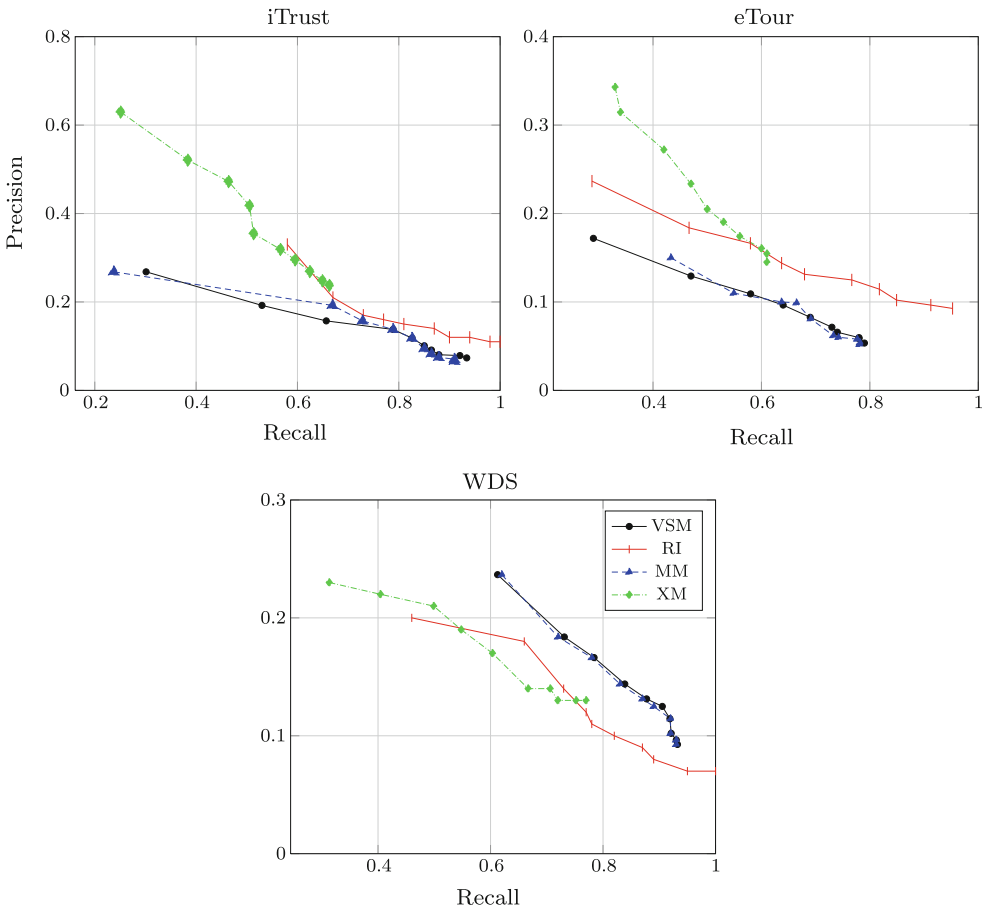

Fig. 7 Performance after applying different refactorings (*RI* Rename Identifier, *MM* Move Method, *XM* eXtract Method)

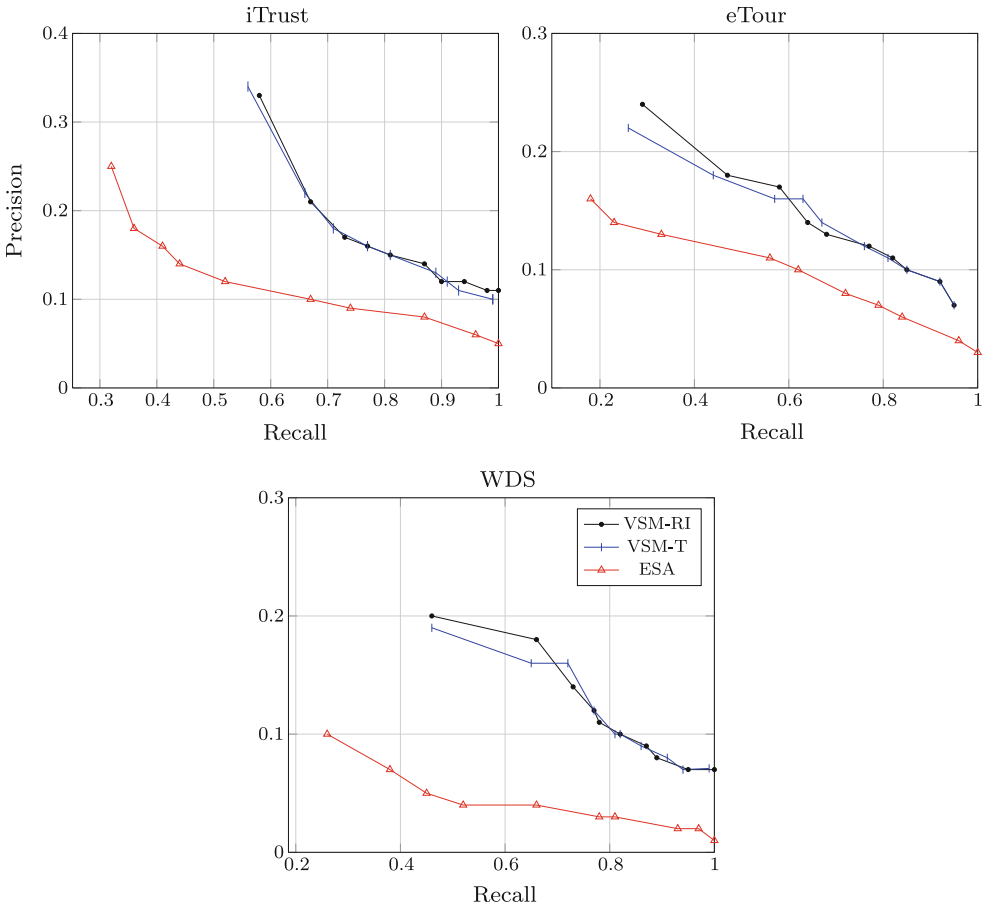

Fig. 8 Comparing the performance of VSM after applying RENAME IDENTIFIER (VSM–RI), with VSM with thesaurus support (VSM-T) and query expansion using explicit semantic analysis (ESA)

on the fact that *Feature Envy* code smell tends to be less dominant and more complicated to detect in software systems than other code smells such as code clones [9, 82, 91]. In fact, further analysis shows that even when a method is moved to another class, it is often still highly referenced (called) in its original class; thus, the track is unlikely to get discontinued, causing MM refactoring to have no obvious impact on the performance. However, it is important to point out that in some cases, where a high density of misleading signs were detected, MM was able to reverse that effect, thus resulting in a slight increase in the recall, especially in *WDS* and *eTour*, however, that improvement was statistically insignificant.

5.5 Discussion

Our analysis has revealed that, in terms of precision and recall, maintaining a domain-specific thesaurus can be equivalent to applying RI refactoring. Therefore, this particular refactoring can be considered as an alternative strategy to handle vocabulary mismatch in software artifacts. However, refactoring provides a more systematic way to handle this problem. In other words, instead of separately maintaining an external ad hoc dictionary of the system's vocabulary and their synonyms, this process can be handled internally through refactoring, as an integral part of the evolution process. As mentioned earlier, RI refactoring is in fact the most applied refactoring in practice, and it has already been integrated in most contemporary IDEs [1, 29, 73, 75].

In terms of effort, as Table 1 shows, the amount of effort required to rename identifiers can be comparable with the external thesaurus technique. While in RI human analysts still have to select new identifiers' names, when building an external thesaurus, synonym relations have to be identified manually. In addition, both methods require a sufficient knowledge of the system's application domain. However, thesaurus-based methods often require calibrating a certain parameter (α in Eq. 7) to achieve the desired performance levels [48], while no calibration or optimization is required when applying RI. In addition, the procedure we propose to identify renaming opportunities helps to alleviate a considerable amount of the effort required to identify misleading signs. In fact, the research on fully

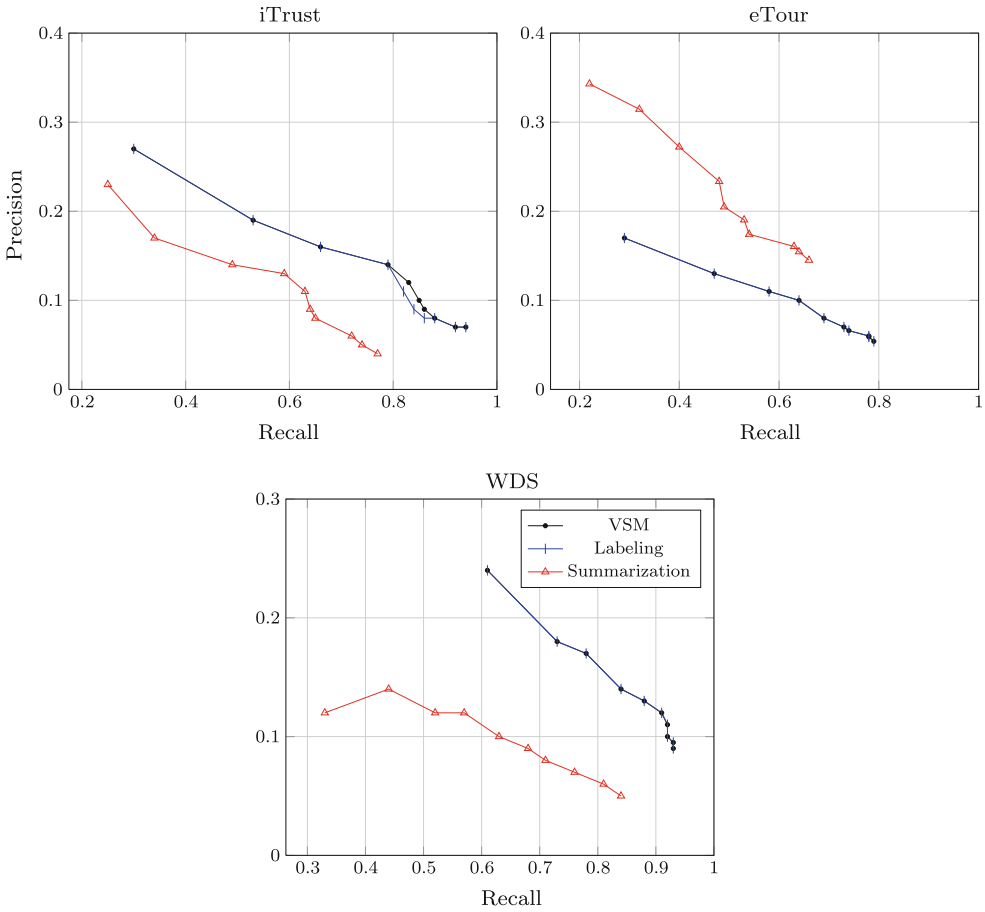

Fig. 9 Comparing the performance of code summarization and code labeling techniques in preserving effective traceability signs

automating this process has noticeably advanced in recent years, especially in the domain of acronyms and abbreviations expansion, opening the door for this process to be fully automated [16, 56, 89].

Our results also show that a simple code labeling technique can fill the vocabulary gap that might result from removing code clones in the system. In terms of effort, code labeling techniques are fully automated, so the human effort is minimized. However, it is important to point out that these implanted labels (signs) are also subject to become outdated as code evolves, thus generating misleading tracks. Therefore, it is important to keep such labels up-to-date and in sync with any changes affecting the code segments they describe.

Finally, even though moving misplaced signs in the system did not result in a statistically significant improvement in the performance, such refactoring can still have an influence on traceability, especially in safety critical systems, where losing even one critical link could be detrimental [18]. However, unlike the renaming process, MM is a nontrivial process, and often results in introducing bugs in the system [75]. Therefore, a careful cost-benefit analysis might be required to determine if performing such transformation is worthwhile.

Our findings in this paper helped in exploring several issues related to applying refactoring as a performance enhancement strategy in IR-based automated tracing. In particular, our study provides insights into developers' actions that might have an impact on the system's traceability during evolution, and reinforced past proposals advocating the use of consistent, and regular vocabulary in identifiers' names [12]. In addition, our analysis revealed how potentially negative effects of removing code clones could be reversed through code labeling, an option that might be important to have in code clone refactoring tools [10, 50].

6 Limitations

This study has several limitations that might affect the validity of the results. Threats to external validity impact the generalizability of results [25]. In particular, the results of this study might not generalize beyond the underlying experimental settings. A major threat to our external validity comes from the datasets used in this experiment. In particular, two of the projects were developed by students and are likely to exhibit different characteristics from

industrial systems. We also note that our traceability datasets are of medium size, which may raise some scalability concerns. Nevertheless, we believe that using three datasets from different domains, including a proprietary software product, helps to mitigate these threats.

Another threat to the external validity might stem from the fact that we only experimented with three refactorings. However, the decision of using these particular refactorings was based on careful analysis of the IR-based automated tracing problem. In addition, these refactorings have been reported to be the most frequently used in practice [67, 92]. Another concern is the fact that only requirements-to-code-class traceability datasets were used. Therefore, our findings might not necessarily apply to other types of traceability such as requirements-to-requirements, requirements-to-design or even different granularity levels such as requirements-to-method. However, our decision to experiment only with requirements-to-class datasets can be justified based on the fact that refactoring has excelled in source code, especially Object-Oriented code, more than any other types of artifacts; thus, we find it appropriate at the current stage of research to consider this particular traceability type at this granularity level.

Other threats to the external validity might stem from specific design decisions such as using VSM with TFIDF weights as our experimental baseline. Refactoring might have a different impact on other IR methods such as LSI and ESA; thus, different results might be obtained. Also, a threat might come from the selection of procedures and tools used to conduct refactoring. However, we believe that using these heavily used and freely available open source tools helps to mitigate this threat. It also makes it possible to independently replicate our results.

Internal validity refers to factors that might affect the causal relations established in the experiment. A major threat to our study's internal validity is the level of automation used when applying different refactorings. In particular, an experimental bias might stem from the fact that the renaming process is a subjective task carried out by the researchers. In addition, human approval of the outcome of the refactoring process was also required. However, as mentioned earlier, in the current state-of-the-art in refactoring research and practice, human intervention is a must [36, 71]. In fact, it can be doubtful whether refactoring can be fully automated without any human intervention [51]. Therefore, these threats are inevitable. However, they can be partially mitigated by automation.

In our experiment, there were minimal threats to construct validity as standard IR measures, which have been used extensively in requirements traceability research, were used to assess the performance of different treatments (recall, precision, MAP and DiffAR). We believe that these two sets of measures sufficiently capture and quantify the different aspects of methods evaluated in this study.

7 Related work

Our work in this paper can be classified under the research categories of enhancing IR-based traceability, and managing traceability in evolving software systems. The former category is concerned with investigating strategies, beyond the underlying retrieval mechanism, that might impact the overall performance of IR-based automated tracing methods, while the latter is focused on strategies to mitigate the risks of software evolution on traceability. In what follows, we review seminal work in these domains, and briefly describe how such work relates to, or can be distinguished from, our work.

7.1 Performance enhancement

Huffman-Hayes et al. [47] used IR with key-phrases and VSM with thesaurus support to recover traceability links between requirements. The former approach associates a list of domain-related technical terms, or key-phrases, with the document repository, while the latter uses a supporting synonym dictionary. Evaluating these techniques revealed that, retrieval with key-phrases improved the recall, but the precision declined. However, VSM with thesaurus support improved both recall and precision. These findings come aligned with our findings in this paper regarding the significant impact of handling the vocabulary mismatch problem in software artifacts. However, our approach uses a systematic and internal approach based on refactoring to handle this problem, rather than maintaining an ad hoc external thesaurus.

In a more recent work by the same authors, relevance feedback from human analysts was incorporated to improve the outcome of IR-based tracing tools [48]. In particular, analysts' link classification decisions were fed back to the tracing tool to help in regenerating more accurate lists of candidate links. Similarly, De Lucia et al. [23] incorporated relevance feedback in the process. However, their approach was incremental in the sense that multiple iterations of link generation were performed, and analysts' feedback was incorporated at each iteration. Evaluating these approaches over multiple datasets showed that using analysts' feedback to tune the weights in the term-by-document matrix of VSM improved the final trace results. In our work, human feedback is also incorporated in the process. However, instead of directly utilizing human classification decisions, the feedback is indirectly incorporated, before the links are generated, through

actions such as renaming identifiers and newly extracted methods.

Under the same category of performance enhancement comes the work of Cleland-Huang et al. [19], who introduced three enhancement strategies to improve the performance of the probabilistic network (PN) model in automated tracing. These strategies included: hierarchical modeling, logical clustering of artifacts, and semi-automated pruning. Results showed that these enhancement strategies could be used effectively to improve trace retrieval results and increase the practicality of tracing tools. Similarly, Niu and Mahmoud [76] proposed an approach based on the cluster hypothesis to improve the quality of candidate link generation for requirements tracing. The main assumption was that correct and incorrect links can be grouped into high-quality and low-quality clusters, respectively. Result accuracy can thus be enhanced by identifying and filtering out low-quality clusters. Since the main objective of refactoring is to enhance the internal quality of software systems, our approach can serve as a preprocessing step for these strategies, improving the underlying structure of the system before applying techniques such as link clustering and hierarchical modeling of artifacts.

In an attempt to overcome the semantic gap between software artifacts in the system, Gibiec et al. [40] used a web-based query expansion algorithm to trace requirements. Similarly, Mahmoud et al. [64] introduced semantic relatedness as an alternative method for query expansion. Thorough evaluation of these different methods showed their ability to achieve higher recall levels, especially in recovering a portion of the *hard-to-trace* requirements. However, as shown in our analysis, this improvement in the recall often comes with a significant decline in the precision. This high ration of false positives usually results from the noise such methods bring from consulting external sources of knowledge for query expansion. In contrast, our approach keeps the noise levels under control, by providing a more focused and more localized method to overcome the same problem.

7.2 Maintaining traceability during evolution

Cleland-Huang et al. [17] presented an event-based approach that establishes traceability links through the use of publish-subscribe relationships between dependent objects in the system. When a significant change to a certain requirement occurs, an event notification message is published to all the subscribed dependent objects. Therefore, ensuring that all these publish-subscribe relations (trace links) are up-to-date or consistent during system evolution. Our work can be distinguished from this work based on the fact that this approach handles the change from the requirement side of the link, while the proposed approach in this paper handles evolution from the source code side. Our approach is based on the observation that code is more prone to change than requirements [11, 59]. Therefore, working on that side of the link is expected to have more immediate effect on traceability.

Egyed [31] proposed an approach that uses observations about the runtime behavior of the system to detect associations among functional scenarios and their executing code. In particular, traces are defined based on the data flow in the form of a footprint graph. A footprint is defined as the lines of code used while executing a scenario. Using our approach, the code does not have to execute or even compile, thus avoiding complications related to the runtime behavior of the system. In addition, no test cases or usage scenarios are needed.

Antoniol et al. [5] proposed an automatic approach to identify class evolution discontinuities due to possible refactorings. The approach identifies links between classes obtained from refactoring, and cases where traceability links were broken due to refactoring. Our approach can be related to this work in the sense that we propose mitigation strategies to overcome problems that may result from certain refactorings (XM). In addition, we also propose the use of refactoring as a preprocessing step to enhance the performance, rather than only dealing with the implications of already applied refactorings.

Mäder et al. [61] proposed a rule-based traceability approach for maintaining traceability relations during evolutionary change. This approach revolves around the monitoring of elementary changes that take place to UML model elements, and updating a preexisting set of traceability relations associated with such changes. This insures that changes in the system's structure will be reflected on traceability, thus keeping such relations up-to-date. However, this approach is restricted to the scope of UML-based, object-oriented (OO), software engineering. In contrast, refactoring is not limited to structural and OO code, and no UML models have to be generated for the system.

The approach proposed by Ben Charrada et al. [11] tackles the problem that we tackle in this paper from a different perspective. In particular, the authors proposed an approach to automatically identifying outdated requirements by analyzing source code changes during evolution to identify the requirements that are likely to be impacted by the change. This approach can be complementary to our approach. While our approach works on the decaying vocabulary structure from the code side, their approach works on the same problem but from the opposite side of the traceability link (the trace query). This will accelerate the process of bridging the textual gap in the system.

Finally, since this paper is based on Gotel and Morris's [43] theoretical approach of IR-based automated tracing,

we find it appropriate here to end our discussion with Gotel's latest views on the field. In their most recent roadmap paper, Gotel et al. [42] identified a number of challenges for implementing effective software and systems traceability. In the set of short-term goals that they specified, they emphasized the need for researchers to focus on mechanisms to mix and match approaches to achieve different cost and quality profiles. The work we presented in this paper is aligned with that goal. In particular, our objective is to add to the current incremental effort of this domain in a way that helps to move forward on the automated tracing roadmap.

8 Conclusions and future work

8.1 Conclusions

In this paper, we explored the effect of applying various refactoring methods on the different performance aspects of IR-based tracing methods. Our main hypothesis is that certain refactorings will help to reestablish the decaying traceability tracks of evolving software systems, thus helping IR methods to recover more accurate lists of candidate links. To test our research hypothesis, we examined the impact of three refactorings on the performance of three requirements-to-code datasets from different application domains. In particular, we identified three main problems associated with IR-based automated tracing including, missing, misplaced, and duplicated signs, and we suggested three refactorings to mitigate these problems. Results showed that restoring textual information in the system's artifacts (RI) had a significantly positive impact on the performance. In contrast, refactorings that remove redundant information (XM) affected the performance negatively. The results also showed that moving information between the system's modules (MM) had no noticeable impact on the performance.

Furthermore, in our analysis, we compared the performance of RI with two other commonly used techniques for handling the vocabulary mismatch problem in software systems. These methods include retrieval enhancement with thesaurus support and query expansion techniques. We analyzed the performance of these techniques, exploited their limitations, and demonstrated how refactoring could address these limitations. In addition, we suggested a sign-preserving technique to mitigate the negative impact of refactoring code clones on traceability. In particular, we proposed two methods for generating source code descriptions including code labeling and summarization, and we analyzed and evaluated their effectiveness in preserving traceability information. Results showed that simple code labeling was more successful than code summarization in preserving traceability signs from getting lost when code clones were refactored. Furthermore, an effective traceability sign analysis was conducted to quantify the effect of different investigated refactorings on the traceability tracks in our experimental systems.

8.2 Future work

The line of work in this paper has opened several research directions to be pursued in our future work. These directions can be described as follows:

- *Refactoring* In our future analysis, we are interested in investigating the effect of other refactorings on traceability. In particular, refactorings that work on the structural information of the system (e.g., Extract Class [34]), and target different granularity levels, will be investigated.
- *IR methods* In our future work, we are interested in exploring the effect of refactoring on other IR methods that are often used in automated tracing, such as (LSI) [65] and (LDA) [7, 13]. These methods work by exploiting hidden (latent) structures in software systems, rather than directly matching keywords in software artifacts; thus, they might have a different response to different refactoring methods.
- *Tool support* In terms of tool support, a working prototype that implements our findings in this paper is in order. A working prototype will allow us to conduct long-term studies that will give us a better understanding of the role of human analysts in the process [77]. In particular, quantifying the potential effort-saving of our approach, its usability, and scope of applicability.
- *Automation* As observed in our experiment, there is still a major effort concern when it comes to refactoring. Humans still play a major role in controlling the refactoring process, starting from approving refactoring candidates captured by code smell detection tools, to applying required refactorings, and verifying the outcome of the process. Therefore, in our future work, we will be exploring refactoring automation strategies that can help to alleviate some of the manual effort in the process.

Acknowledgments We would like to thank the partner company for the generous support of our research. This work is supported in part by the US NSF (National Science Foundation) Grant No. CCF1238336.

References

1. Advani D, Hassoun Y, Counsell S (2005) Refactoring trends across N versions of N Java open source systems: an empirical study. SCSIS-Birkbeck, University of London Technical Report

2. Anquetil N, Fourrier C, Lethbridge T (1999) Experiments with clustering as a software remodularization method. In: Working conference on reverse engineering, pp 235–255
3. Anquetil N, Lethbridge T (1998) Assessing the relevance of identifier names in a legacy software system. In: Conference of the centre for advanced studies on collaborative research, pp 4–14
4. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28:970–983
5. Antoniol G, Di Penta M, Merlo E (2004) An automatic approach to identify class evolution discontinuities. In: International workshop on principles of software evolution, pp 31–40
6. Aslam J, Yilmaz E, Pavlu V (2005) A geometric interpretation of *r*-precision and its correlation with average precision. In: Annual international ACM SIGIR conference on research and development in information retrieval, pp 573–574
7. Asuncion H, Asuncion A, Taylor R (2010) Software traceability with topic modeling. In: International conference on software engineering, pp 95–104
8. Aversano L, Cerulo L, Di Penta M (2010) How clones are maintained: an empirical study. In: European conference on software maintenance and reengineering, pp 81–90
9. Baker B (1995) On finding duplication and near-duplication in large software systems. In: Working conference on reverse engineering, pp 86–95
10. Baxter I, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: ICSM, pp 368–377
11. Ben Charrada E, Koziolek A, Glinz M (2012) Identifying outdated requirements based on source code changes. In: International requirements engineering conference, pp 61 –70
12. Binkley D, Lawrie D, Maex S, Morrell C (2009) Identifier length and limited programmer memory. Sci Comput Program 74(7):430–445
13. Blei D, Ng A, Jordan MI (2003) Allocation. J Mach Learn Res 3:993–1022
14. Bourquin F, Keller R (2007) High-impact refactoring based on architecture violations. In: European conference on software maintenance and reengineering, pp 149–158
15. Bruntink M, Van Deursen A, Van Engelen R, Tourwé T (2005) On the use of clone detection for identifying crosscutting concern coden. IEEE Trans Softw Eng 31:804–818
16. Caprile B, Tonella P (2000) Restructuring program identifier names. In: International conference on software maintenance, pp 97–107
17. Cleland-Huang J, Chang C, Christensen M (2003) Event-based traceability for managing evolutionary change. IEEE Trans Softw Eng 29(9):796–810
18. Cleland-Huang J, Heimdahl M, Huffman-Hayes J, Lutz R, Mäder P (2012) Trace queries for safety requirements in high assurance systems. In: International conference on requirements engineering: foundation for software quality, pp 179–193
19. Cleland-Huang J, Settimi R, Duan C, Zou X (2005) Utilizing supporting evidence to improve dynamic requirements traceability. In: International conference on requirements engineering, pp 135–144
20. Cleland-Huang J, Settimi R, Romanova E (2007) Best practices for automated traceability. Computer 40(6):27–35
21. David K (2003) Selected papers on computer languages. In: CSLI lecture notes, vol 139. Center for the Study of Language and Information
22. De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichelle S (2012) Using IR methods for labeling source code artifacts: Is it worthwhile? In: International conference on program comprehension, pp 193–202
23. De Lucia A, Oliveto R, Sgueglia P (2006) Incremental approach and user feedbacks: a silver bullet for traceability recovery. In: International conference on software maintenance, pp 299–309
24. De Lucia A, Oliveto R, Tortora G (2009) Assessing IR-based traceability recovery tools through controlled experiments. Empir Softw Eng 14(1):57–92
25. Dean A, Voss D (1999) Design and analysis of experiments. Springer, New York
26. Deerwester S, Dumais S, Furnas G, Landauer T, Harshman R (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407
27. Deissenböck F, Pizka M (2005) Concise and consistent naming. In: International workshop on program comprehension, pp 97–106
28. Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30
29. Dig D, Johnson R (2005) The role of refactorings in API evolution. In: International conference on software maintenance, pp 389–398
30. DualaEkoko E, Robillard M (2010) Clone region descriptors: representing and tracking duplication in source code. ACM Trans Softw Eng Methodol 20(1):1–31
31. Egyed A (2003) A scenario-driven approach to trace dependency analysis. IEEE Trans Softw Eng 9(2):116–132
32. Eick S, Graves T, Karr A, Marron J, Mockus A (1998) Does code decay? Assessing the evidence from change management data. IEEE Trans Softw Eng 27(1):1–12
33. Feilkas M, Ratiu D, Jurgens E (2009) The loss of architectural knowledge during system evolution: an industrial case study. In: International conference on program comprehension, pp 188–197
34. Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A (2012) Identification and application of extract class refactorings in object-oriented systems. J Syst Softw 85(10):2241–2260
35. Fontanaa F, Braionea P, Zanonia M (2011) Automatic detection of bad smells in code: an experimental assessment. J Object Technol 11(2):1–8
36. Fowler M (1999) Refactoring: improving the design of existing code. Addison–Wesley, Reading
37. Furnas G, Deerwester S, Dumais S, Landauer T, Xarshman R, Streeter L, Lochbaum K (1988) Information retrieval using a singular value decomposition model of latent semantic structure. In: Annual international ACM SIGIR conference on research and development in information retrieval, pp 465–480
38. Gabrilovich E, Markovitch S (2007) Computing semantic relatedness using wikipedia-based explicit semantic analysis. In: international joint conference on artificial intelligence, pp 1606–1611
39. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
40. Gibiec M, Czauderna A, Cleland-Huang J (2010) Towards mining replacement queries for hard-to-retrieve traces. In: International conference on automated software engineering, pp 245–254
41. Giulio A, Caprile B, PotrichA Tonella P (2000) Design-code traceability for object-oriented systems. Ann Softw Eng 9(1–4):35–58
42. Gotel O, Cleland-Huang J, Huffman-Hayes J, Zisman A, Egyed A, Grünbacher P, Antoniol G (2012) The quest for ubiquity: a roadmap for software and systems traceability research. In: international conference on requirements engineering, pp 71–80
43. Gotel O, Morris S (2011) Out of the labyrinth: leveraging other disciplines for requirements traceability. In: IEEE international requirements engineering conference, pp 121–130

44. Guerrouj L (2013) Normalizing source code vocabulary to support program comprehension and software quality. In: International conference on software engineering, pp 1385–1388
45. Haiduc S, Aponte J, Moreno L, Marcus A (2010) On the use of automated text summarization techniques for summarizing source code. In: Working conference on reverse engineering, pp 35–44
46. Han E, Karypis G (2000) Centroid-based document classification: analysis and experimental results. In: European conference on principles of data mining and knowledge discovery, pp 424–431
47. Huffman-Hayes J, Dekhtyar A, Osborne (2003) J Improving requirements tracing via information retrieval. In: International conference on requirements engineering, pp 138–147
48. Huffman-Hayes J, Dekhtyar A, Sundaram S (2006) Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans Softw Eng 32(1):4–19
49. Jones K (2007) Automatic summarising: the state of the art. Inf Process Manag 43(6):1449–1481
50. Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670
51. Katić M, Fertalj K (2009) Towards an appropriate software refactoring tool support. In: WSEAS international conference on applied computer science, pp 140–145
52. Kim M, Bergman L, Lau T, Notkin D (2004) An ethnographic study of copy and paste programming practices in OOPL. In: International symposium on empirical software engineering, pp 83–92
53. Kolb R, Muthig D, Patzke T, Yamauchi K (2006) Refactoring a legacy component for reuse in a software product line: a case study: practice articles. J Softw Maint Evol 18(2):109–132
54. Koschke R, Falke R, Frenzel P (2006) Clone detection using abstract syntax suffix trees. In: Working conference on reverse engineering, pp 253–262
55. Laitinen K (1996) Estimating understandability of software documents. SIGSOFT Softw Eng Notes 21(4):81–92
56. Lawrie D, Binkley D, Morrell C (2010) Normalizing source code vocabulary. In: Working conference on reverse engineering, pp 3–12
57. Lawrie D, Feild H, Binkley D (2007) Extracting meaning from abbreviated identifiers. In: International working conference on source code analysis and manipulation, pp 213–222
58. Lehman M (1984) On understanding laws, evolution, and conservation in the large-program life cycle. J Syst Softw 1(3):213–221
59. Lethbridge T, Singer J, Forward A (2003) How software engineers use documentation: the state of the practice. IEEE Softw 20(6):35–39
60. Luo J, Meng B, Liu M, Tu X, Zhang K (2012) Query expansion using explicit semantic analysis. In: International conference on internet multimedia computing and service, pp 123–126
61. Mäder P, Gotel O, Philippow I (2008) Rule-based maintenance of post-requirements traceability relations. In: International requirements engineering conference, pp 23–32
62. Mahmoud A, Niu N (2011) Source code indexing for automated tracing. In: International workshop on traceability in emerging forms of software engineering, pp 3–9
63. Mahmoud A, Niu N (2013) Supporting requirements traceability through refactoring. In: International requirements engineering conference, pp 32–41
64. Mahmoud A, Niu N, Xu S (2012) A semantic relatedness approach for traceability link recovery. In: International conference on program comprehension, pp 183–192
65. Maletic J, Marcus A (2000) Using latent semantic analysis to identify similarities in source code to support program understanding. In: International conference on tools with artificial intelligence, pp 46–53
66. Manning C, Raghavan P, Schtze H (2008) Introduction to information retrieval. Cambridge University Press, Cambridge
67. Mäntylä M, Lassenius C (2006) Drivers for software refactoring decisions. In: International symposium on empirical software engineering, pp 297–306
68. Mayrand J, Leblanc C, Merlo E (1996) Experiment on the automatic detection of function clones in a software system using metrics. In: International conference on software maintenance, pp 244–253
69. Mealy E, Carrington D, Strooper P, Wyeth P (2007) Improving usability of software refactoring tools. In: Australian software engineering conference, pp 307–318
70. Meneely A, Smith B, Williams L (2012) iTrust electronic health care system: a case study, chap. software and systems traceability. Springer, New York
71. Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30(2):126–139
72. Moser R, Sillitti A, Abrahamsson P, Succi G (2006) Does refactoring improve reusability? In: International conference on reuse of off-the-shelf components, pp 287–297
73. Murphy G, Kersten M, Findlater L (2006) How are java software developers using the eclipse IDE. IEEE Softw 23(4):76–83
74. Murphy-Hill E, Black AP (2008) Breaking the barriers to successful refactoring: observations and tools for extract method. In: ICSE, pp 421–430
75. Murphy-Hill E, Parnin C, Black AP (2009) How we refactor and how we know it. In: International conference on software engineering, pp 287–297
76. Niu N, Mahmoud A (2012) Enhancing candidate link generation for requirements tracing: the cluster hypothesis revisited. In: IEEE International requirements engineering conference, pp 81–90
77. Niu N, Mahmoud A, Chen Z, Bradshaw G (2013) Departures from optimality: understanding human analysts information foraging in assisted requirements tracing. In: International conference on software engineering, pp 572–581
78. Opdyke W (1992) Refactoring object-oriented frameworks. Doctoral thesis, Department of Computer Science, University of Illinois at Urbana-Champaign
79. Opdyke W, Johnson R (1990) Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In: Symposium on object-oriented programming emphasizing practical applications
80. Porter M (1997) An algorithm for suffix stripping. In: Readings in information retrieval. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 313–316
81. Roy C, Cordy J (2007) A survey on software clone detection research. Technical report 541. School of Computing TR 2007-541, Queens University
82. Roy C, Cordy J (2008) An empirical study of function clones in open source software. In: Working conference on reverse engineering, pp 81–90
83. Spanoudakis G, Zisman A (2004) Software traceability: a roadmap. Handb Softw Eng Knowl Eng 3:395–428
84. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: International conference on automated software engineering, pp 43–52
85. Sultanov H, Huffman-Hayes J, Kong W (2011) Application of swarm techniques to requirements tracing. Requir Eng J 16(3):209–226
86. Sundaram S, Huffman-Hayes J, Dekhtyar A, Holbrook E (2010) Assessing traceability of software engineering artifacts. Requir Eng J 15(3):313–335
87. Takang A, Grubb P, Macredie R (1996) The effects of comments and identifier names on program comprehensibility: an experimental investigation. J Program Lang 4(3):143–167

88. Teufel S (2007) An overview of evaluation methods in TREC ad hoc information retrieval and TREC question answering. In: Dybkjaer L, Hemsen H, Minker W (eds) Evaluation of text and speech systems. Springer, Netherlands, pp 163–186
89. Thies A, Roth C (2010) Recommending rename refactorings. In: International workshop on recommendation systems for software engineering, pp 1–5
90. Tourwé T, Mens T (2003) Identifying refactoring opportunities using logic meta programming. In: European conference on software maintenance and reengineering, pp 91–100
91. Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Trans Softw Eng 35(3):347–367
92. Wilking D, Kahn U, Kowalewski S (2007) An empirical evaluation of refactoring. e-Inf Softw Eng J 1(1):44–60