Supporting Requirements Traceability through Refactoring

Anas Mahmoud
Computer Science and Engineering
Mississippi State University
Mississippi State, MS, USA
amm560@msstate.edu

Nan Niu
Computer Science and Engineering
Mississippi State University
Mississippi State, MS, USA
niu@cse.msstate.edu

***Abstract*—Modern traceability tools employ information retrieval (IR) methods to generate candidate traceability links. These methods track textual signs embedded in the system to establish relationships between software artifacts. However, as software systems evolve, new and inconsistent terminology finds its way into the system's taxonomy, thus corrupting its lexical structure and distorting its traceability tracks. In this paper, we argue that the distorted lexical tracks of the 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 this novel hypothesis, we investigate the effect of integrating various types of refactoring on the performance of requirements-to-code automated tracing methods. In particular, we identify the problems of missing, misplaced, and duplicated signs in software artifacts, and then examine to what extent refactorings that restore, move, and remove textual information can overcome these problems respectively. We conduct our experimental analysis using three datasets from different application domains. Results show that restoring textual information in the system has a positive impact on tracing. In contrast, refactorings that remove redundant information impact tracing negatively. Refactorings that move information among the system modules are found to have no significant effect. Our findings address several issues related to code and requirements evolution, as well as refactoring as a mechanism to enhance the practicality of automated tracing tools.**

***Index Terms*—Refactoring, traceability, information retrieval.**

I. Introduction

Traceability is defined as *"the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)"* [1]. Establishing and maintaining traceability information is vital to several software engineering activities such as program comprehension [2], Verification and Validation (V&V) [3], [4], impact analysis [5], feature location [6], and software reuse [7]. Traceability is often accomplished in practice by linking various software artifacts (e.g., requirements, design, source code, and test cases) through a matrix, called a traceability matrix (TM). However, when dealing with large-scale complex systems in which software artifacts evolve constantly over time, building the TM manually can become a tedious, exhaustive, and error-prone task [1], [3], [8], [9].

To reduce the effort associated with the manual approach, modern traceability tools employ information retrieval (IR) methods for automated support [3], [9], [10]. Examples of IR methods that have been heavily investigated in the automated tracing literature include: Vector Space Model (VSM) [3], Latent Semantic Indexing (LSI) [11], and Probabilistic Network Model (PN) [9]. These 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 the same words are used whenever a particular concept is described [12], [13]. Therefore, artifacts having a high textual similarity probably share several concepts, so they are likely good candidates to be traced from one another [12]. However, as projects evolve, new and inconsistent terminology gradually finds its way into the system's taxonomy [14], causing topically related system artifacts to exhibit a large degree of variance in their lexical contents [15], [16]. This phenomena is known as the *vocabulary mismatch* problem and is regarded as one of the principal causes of poor accuracy in retrieval engines [17].

When lexical tracks in the system get distorted, cases such as hard-to-trace, or stubborn, requirements emerge [18]. A potential solution for this problem is to systematically recover the decaying lexical structure of the system. We believe this can be achieved through refactoring, a set of behavior-preserving transformations to improve the quality of a software system without changing its external behavior [19]. These transformations act on the non-formal information of software artifacts, namely the features that do not have an influence on the functionality of the system. Such information is embedded in the taxonomy of the source code and is used by tractability tools to generate candidate links. Based on that common ground, in this paper, we hypothesize that certain refactorings will help to re-establish the system's corrupt lexical structure, thus improving the retrieval capabilities of IR methods working 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 refactorings should be integrated? Which techniques have more influence on the system's traceability? And how to evaluate

978-1-4673-5765-4/13 © 2013 IEEE

such influence? To answer these questions, in this paper, we propose a framework for integrating refactoring techniques in the automated tracing process. We conduct an experiment using three datasets from various application domains to build, calibrate, and evaluate our framework. Our main objective is to describe a novel, yet a practical and cost-effective approach, for systematically enhancing the performance of automated tracing tools.

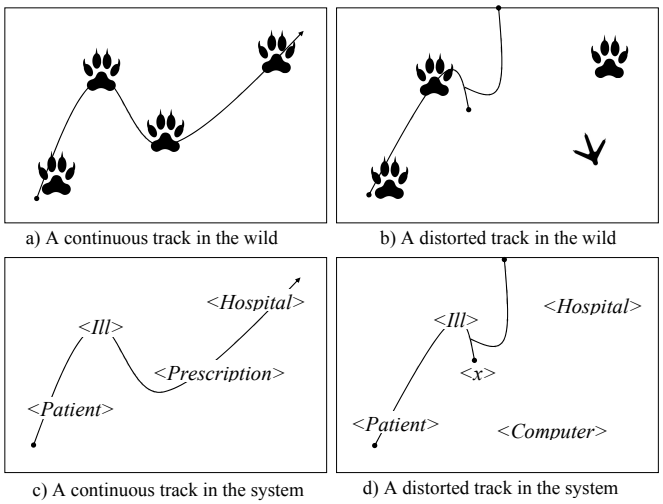
The rest of the paper is organized as follows. Section II presents a theoretical foundation of IR-based automated tracing. Section III introduces refactorings and describes the different categories of refactoring used in our analysis. Section IV describes our research methodology and experimental analysis. Section V presents the results and the study limitations. Section VI discuses our findings and their implications. Section VII reviews related work. Finally, Section VIII concludes the paper and discusses future work.

II. IR-Based Automated Tracing

To understand the mechanisms of automated tracing tools, we refer to the main theory underlying IR-based automated tracing. In their vision paper, Gotel and Morris [13] 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. Fig. 1-a 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 concept, left by a software developer or a system engineer. Fig. 1-c shows a continuous track of related words from the health care domain <*patient, ill, prescription, hospital*>. The task of the 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 [14], [20]. Next 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. Fig. 1-d 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. 1-b).
- 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. 1-d. 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. 1-b shows a bird's footprint left on the mammal's track in Fig. 1-a).
- 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 [21]. In our example in Fig. 1-d, 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. 1-c. Some animals adopt this strategy in the wild to confuse their predators *i.e.* they duplicate their footprints in different directions at different periods of time.



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

Fig. 1. Illustration of sign tracking.

Our conjecture in this paper is that refactoring will help to reverse the effect of these symptoms, thus systematically re-establishing track in the system. Following is a brief description of refactoring, its operation, and its main categories.

III. Refactoring

The main goal of refactoring is to improve the quality of a software system (e.g., increase maintainability, reusability, and understandability) by systematically applying a set of behavior-preserving transformations that do not alter the external behavior of the system, yet improve its internal structure [19]. While such transformations can be applied to various types of artifacts, such as design and requirements, they are mostly known for affecting source code [22]. Program refactoring starts by identifying *bad smells* in source code. Bad smells are *"structures in the code that suggest the possibility of refactoring"* [23]. 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 [24]. A comprehensive catalog of code refactorings can be found in 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 (bad smell), 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 a repetitive, time-consuming, and error-prone [25]. The semi-automated approach is what most contemporary integrated development environments (IDE's) 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 smell 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 automated process is left to the human [26].

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 improve it [19]. 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 characteristic of the problem, the cost-benefit analysis, or the goal of refactoring (e.g., improving robustness, extensibility, reusability, understandability, or performance of the system) [22], [25]. In automated tracing, the main goal of adopting refactoring is to improve the system's lexical 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 non-formal information of the system: As mentioned earlier, IR-based tracing methods exploits non-formal information of software artifacts [27]. Based on that, for any refactoring to have an impact on automated tracing methods, it should directly affect the system's textual content.
- Coverage: Traceability links can be spreading all over the system, linking a large number of the system's artifacts through various traceability relations [7]. Therefore, any adopted refactoring shall affect as many software entities as possible. Refactorings affecting only a few entities are unlikely to have a significant impact on the performance.
- 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 [28]. Automating these two steps will help to alleviate a large portion of effort usually associated with manual refactoring.

Based on these requirements, we identify three categories of refactorings that can be integrated in the automated tracing process. These categories include: refactorings that restore, remove and move textual information in the the system. Following is a description of these categories.

A. Restore Information

Refactorings under this category target the degrading taxonomy in the system. The main goal is to restore the textual knowledge that is lost over iterations of system evolution. Several refactorings can be classified under this category (e.g. Table I). The most popular refactoring in this category is *Rename Identifier* (RI). As the name implies, this transformation includes simply renaming an identifier (e.g. variable, class, structure, method, or field) to give it a more relevant name [23]. This particular refactoring is expected to target the *Missing Sign* problem affecting traceability methods. As mentioned earlier, to be considered in our analysis, refactorings should provide support for automatic detection of code smell. To identify opportunities for *Rename Identifier* refactoring, we apply the following rules:

1) Identifiers with less than 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. If any of these parameters also appears in the requirements being traced it is also expanded to ensure an exact match.
2) Identifiers which have a special word as part of their names: For example, the variable *PnString* is expanded to *PatientNameString*.
3) Identifiers with generic names: For example, the function *import* is renamed to indicate what exactly it imports. In our health care system *import* is expanded to *importPatientRecords*.

Once the candidate identifiers for renaming have been identified, we use the refactoring tool available in Eclipse 4.2.1 IDE to carry out the refactoring process. This will ensure that all corresponding references in the code are updated automatically. Finally, the code is compiled to make sure no bugs are introduced during refactoring.

B. Move Information

This category of refactorings 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 [29]. Refactorings under this category provide a remedy against the *Feature Envy* bad smell. An entity has *Feature Envy* when it uses, or being used by, the features of a class other than its own (different from that where it is declared). This may indicate that the entity is misplaced [23]. Examples of refactorings under this category are shown in Table I.

In our experiment, we adopt *Move Method* (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 [30], in which they

TABLE I
REFACTORING CATEGORIES WITH SAMPLE REFACTORING

Restore Information	Move Information	Remove Information
Rename Identifier	Move Method	Extract Method
Add Parameter	Move Parameter	Decompose Conditional
Split Temporary Var -riable	Push Down Field	Parameterize Method
	Push Down Method	Remove Dou- ble Negative

introduced a novel entity placement metric to quantify how well entities have been placed in code. This semi-automatic strategy starts by identifying a set of the entities each method accesses (parameters or other methods). *Feature Envy* bad smell is then detected by measuring the strength of coupling that a method has to methods (or data) 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 methodology has been implemented as an Eclipse plug-in that identifies *Feature Envy* bad smells and allows the user to apply the refactorings that resolve them [31]. *Move Method* refactoring of Eclipse 4.2.1 IDE is used to move methods. 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 [30].

C. Remove Information

These refactorings remove redundant or unnecessary code in the system. Table I shows examples of refactorings that can be classified under this category. A popular smell such refactorings often handle is *Duplicated Code*. This smell, usually produced by Copy-and-Paste programming [32], 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 [33]. Exact duplicated code structures can be detected by comparing text [23]. 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 [34].

The most frequent way to handle code duplicates is *Extract Method* (EM) refactoring [35], [36]. 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. In case of a tie, the method is assigned randomly. Finally, the extracted method is given a name based on the context. By removing potentially ambiguous duplicates this transformation is expected to target the *Duplicated Sign* problem of software artifacts. We use Duplicated code detection tool (SDD) [37], an Eclipse plug-in to detect the *Duplicated Code* instances. *Extract Method* refactoring available in Eclipse 4.2.1 IDE is used to apply the transformations.

TABLE II
EXPERIMENTAL DATASETS

Dataset	LOC	COM	No. Req.	No. SC	Links
iTrust	20.7K	9.6K	50	299	314
eTour	17.5K	7.5K	58	116	394
WDS	44.6K	10.7K	26	521	229

IV. METHODOLOGY AND RESEARCH HYPOTHESIS

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

A. Datasets

Three datasets are used to conduct the experiment in this paper including: *iTrust*, *eTour*, and *WDS*. Next 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 [38]. 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.
- *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.
- *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 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 II 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 e.g., number of requirements (No. Req.) and number of code elements (No. SC), and the number of correct traceability links.

B. Experimental Analysis

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

- *Refactoring*: Initially the system is refactored using various refactorings mentioned earlier. The goal is to improve

the system lexical structure before indexing. In all of our experimental datasets, traceability links are established at class granularity level (e.g. requirement-to-class) [3]. This limits our analysis in this paper to refactorings that work within the class scope (e.g. *Move Method* and *Extract Method*), 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.

Results of applying different refactorings over our datasets are shown in Table III. The table shows number of affected entities (E) in each dataset, (e.g number of moved or extracted methods and number of renamed identifiers), the number of affected classes in each system (C), and the number of affected classes in the gold-standard, or classes being traced (C').

- *Indexing*: This process starts by extracting textual content (e.g. comments, code identifiers, requirements text) from input artifacts. Lexical processing (e.g. splitting code identifiers into their constituent words) is then applied. Stemming is performed to reduce words to their roots (i.e., reducing a word to its inflectional root: *"patients"* $\rightarrow$ *"patient"*). In our analysis we use Porter stemming algorithm [39]. 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 [40].
- *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 [3]. 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 [3], [41].
- *Evaluation*: At this step, different evaluation metrics are used to assess the different aspects of the performance. In particular, two categories of performance measures are used to evaluate the quality and the browsability of the generated lists of candidate links. The following is a description of these categories.

C. Metrics

Sundaram *et al.* [42] 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:

1) Quality Measures: Precision (*P*) and Recall (*R*) are the standard IR metrics 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 [43]. Formally,

TABLE III
AFFECTED ENTITIES (E) AND CLASSES (C, C') BY REFACTORING

	iTrust			eTour			WDS		
Refactoring	E	C	C'	E	C	C'	E	C	C'
RI	175	113	110	85	63	57	203	174	166
MM	22	44	44	17	31	29	24	62	61
EM	132	201	193	45	92	88	62	102	98

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:

$$\textit{R (Recall)} = |A \cap B|/|A| \tag{1}$$

$$\textit{P (Precision)} = |A \cap B|/|B| \tag{2}$$

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 give any information about the list browsability. To reflect such information, other metrics 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 $L = \{(d,h)|sim(d,h)\}$ be a set of candidate traceability links generated by the tool, where $sim(d,h)$ is the similarity between *d* and *h* score given by tool. 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'). Secondary metrics can be described as:

- Mean Average Precision (MAP): is a measure of quality across recall levels [44]. It can be described as the mean precision after each relevant link retrieved (true postive). Eq. 3 describes *MAP*. A method or a tool that produces a higher *MAP* is superior.

$$\textit{MAP} = \frac{1}{|H|}\sum_{j=1}^{H}\frac{1}{m_j}\sum_{k=1}^{m_j} Precion(L_{jT}) \tag{3}$$

- DiffAR: measures the contrast of the list [45]. 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. Eq. 4 defines *DiffAR*.

$$\textit{DiffAR} = \frac{\sum_{(h,d)} sim(h,d)}{|L_T|} - \frac{\sum_{(h',d')} sim(h',d')}{|L_F|} \tag{4}$$

Performance of each dataset after applying a certain refactorings, in comparison to the baseline (VSM), is presented as a precious/recall curve over various threshold levels ($< .1, .2, ..., 1 >$) [3]. 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 non-parametric test that makes no assumptions about the distribution of the

data [46]. This test is applied over the combined samples from two related samples or repeated measurements on a single sample (before and after effect). IBM SPSS Statistics software package is used to conduct the analysis. We use $\alpha = 0.05$ to test the significance of the results. Note that different refactorings are applied independently, so there is no interaction effect between them.

V. Results and Impact

Fig. 2 shows the recall/precision data of our three datasets after applying the different refactorings (RI: *Rename Identifier*, MM: *Move Method* and EM: *Extract Method*) in comparison to the baseline. Analysis of variance over the results is shown in Table IV. In general, the results show that different types of refactorings vary in their impact on the performance. In details, *Rename Identifier* refactoring has the most obvious positive impact on the results, affecting the recall significantly in all three datasets. In *iTrust* dataset, both precision and recall have improved significantly, achieving optimal recall levels at higher thresholds. The same performance is detected in *eTour* dataset in which the improvement in the recall and the precision over the baseline is statistically significant. In *WDS* dataset the precision has dropped significantly with the significant increase in the recall. This can be explained based on the inherent traded-off between precision and recall; even though renaming identifiers has helped to retrieve more true positives, it also retrieve a high number of false positive.

The results also show that *Move Method* refactoring has the least influence on the performance. In all datasets no significant improvements in the recall or the precision are detected. In fact, the performance after applying this particular refactoring is almost equivalent to the baseline. In contrast, statistical analysis show that *Extract Method* refactoring has an overall negative impact on the performance. In terms of recall, the results show that removing redundant textual knowledge from the system has caused a significant drop in the coverage, taking the recall down to significantly low levels in all three datasets. The significant increase in the precision levels can be simply explained based on the inherent trade-off between precision and recall i.e. smaller numbers of links are retrieved, thus, the precision has improved.

In terms of browsability, statistical analysis results in Table IV show that *Rename Identifier* and *Move Method* have no significant impact on the average DiffAR. However, *Extract Method* seems to be achieving significantly better performance over the baseline. In terms of *MAP*, Fig. 3 shows the superior performance of *Extract Method* over other refactorings in comparison to the baseline. However, this behavior is expected based on the fact that *VSM* retrieves the smallest number of links after applying *Extract Method*, hence, achieves the highest precision values, which in turn results in higher MAP values. MAP results also show the inconstant performance of *Rename Identifier* across the different datasets. In *iTrust*, *Rename Identifier* starts with good MAP values at lower threshold levels, but could not beat the baseline halfway through; no significant difference in the performance is detected. In contrast, in *eTour* it achieves significantly better performance than the baseline and significantly worst performance in *WDS*. Finally, *Move Method* 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.

In general, our results suggest that *Rename Identifier* refactoring has the most significant positive effect on the results, improving the recall to significantly higher levels in all datasets. In contrast, *Extract Method* had a significantly negative impact, taking the recall down to significantly lower levels in all three datasets, and *Move Method* has no clear impact on the performance. Automated tracing methods emphasize recall over precision [3]. 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 *Rename Identifier* refactoring has the most potential as a performance enhancement technique for IR-based automated tracing.

A. Limitations

This study has several limitations that might affect the validity of the results. Threats to external validity impact the generalizability of results [47]. 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 are 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 [35], [36]. 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, requirement-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.

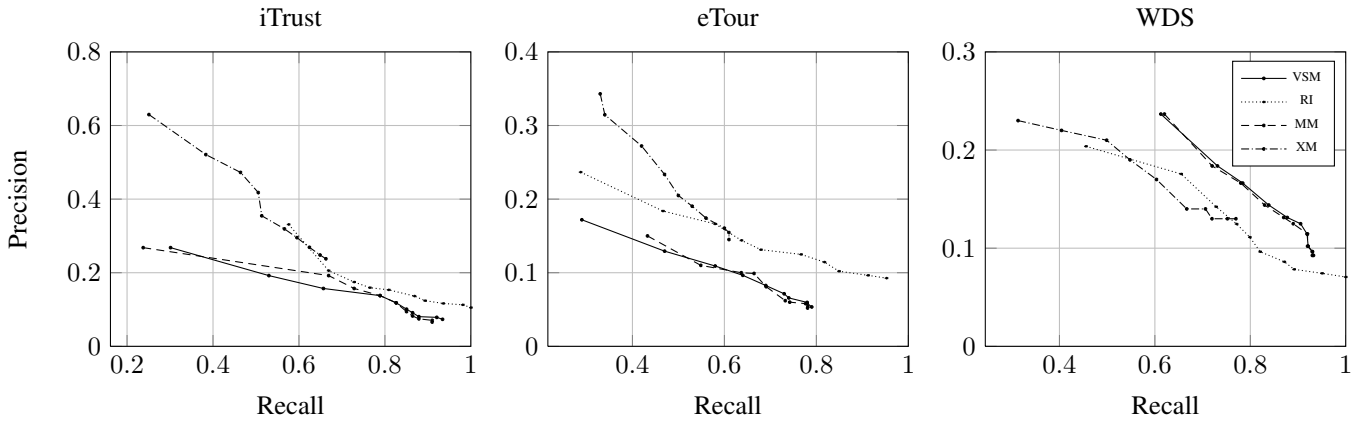


Fig. 2. Performance after applying different refactorings (RI: Rename Identifier, MM: Move Method, XM: eXtract Method)

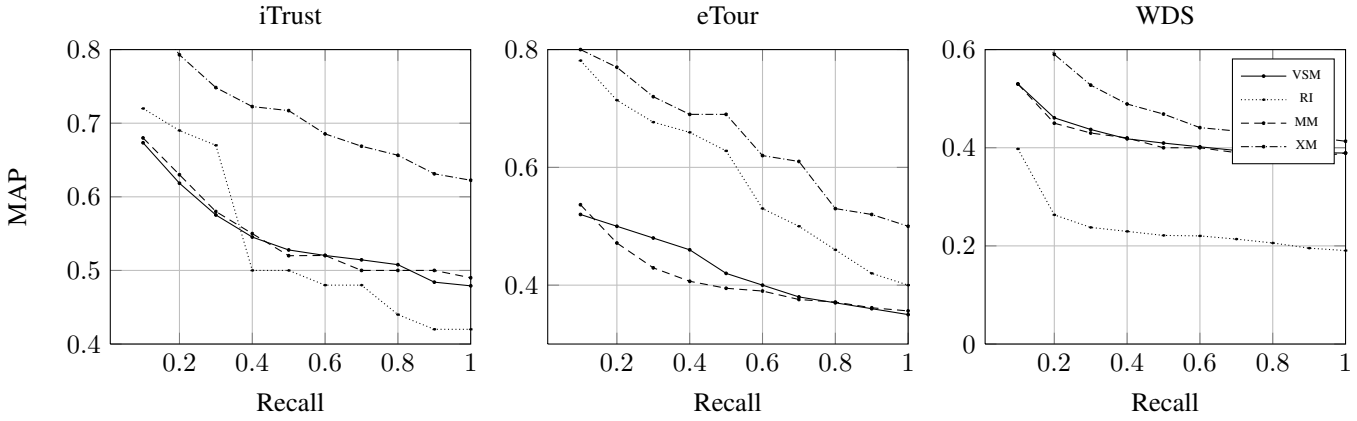


Fig. 3. Performance after applying different refactorings (RI: Rename Identifier, MM: Move Method, XM: eXtract Method)

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 [22], [23]. In fact, it can be doubtful whether refactoring can be fully automated without any human intervention [26]. 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 (recall, precision, MAP and DiffAR), which have been used extensively in requirements traceability research, were used to assess the performance of different treatments applied. We believe that these two sets of measures sufficiently capture and quantity the different aspects of methods evaluated in this study.

VI. Discussion

Essential questions to answer when refactoring a software system are what refactorings to apply to certain situations and how to apply them [23], [26]. In this paper we tried to address such questions from a traceability perspective. In particular, our findings in this paper provide insights into developers' actions that might have an impact on the system's traceability during evolution, the extent of such impact, best ways to implement beneficial transformations, and how potentially negative effects can be reversed.

Our results suggest that restoring textual information has the most positive impact on the system's traceability. In particular, *Rename Identifier* refactoring targets the *vocabulary mismatch* problem, which seems to be the most dominant problem affecting traceability tools. In the automated tracing

TABLE IV
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, .017)	(-2.803, .005)	(-2.701, .007)	(-2.803, .005)	(-2.090, .037)	(-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, .017)	(-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, .004)	(-2.809, .005)	(-2.803, .005)	(-2.803, .005)	(-3.051, .002)

literature, this particular transformation can be considered equivalent to other techniques that are usually used to handle the vocabulary mismatch problem. Such techniques include directly implanting tracing information in the system or using external thesaurus [3], and query expansion [18], [41]. While our approach confirms the benefits of such techniques, it suggests a more systematic way for implementing them. In particular, instead of creating and maintaining separate *ad hoc* traceability thesaurus, arbitrarily implanting textual signs in the system, or using external resources to expand the trace query, this process can be handled systematically through refactoring. This can be particularly beneficial to approaches that use patterns in the namesof program entities to recover traceability links in software systems [48]. Refactoring is now being advocated as an essential step in any software process. For example, in agile methods, refactoring has already been integrated as a regular practice in the software life cycle [22]. In addition, refactoring tools, which support a large variety of programming languages, have been integrated into most popular development environments [26].

A surprising observation in this paper is that removing redundant information from the system has a negative impact on the performance. This suggests that redundant information, while it is often considered a bad code smell from a refactoring perspective [49], is actually serving a positive purpose for traceability link recovery. Such observation can be used to alter existing refactorings to mitigate the negative impact they might have on traceability. For example, a suggested treatment to reverse the negative effect of *Extract Method* is to use comments. Whenever redundant code (code clone) is removed, appropriate comments that describe the removed code can be automatically inserted to fill the textual gap left by removing code duplicates. This can be achieved by utilizing automatic summarising techniques to generate comments from source code [50].

Our results also show that moving information among the system's modules has no significant impact on traceability, which suggests that misplaced signs are not as problematic for traceability tools as missing or duplicated signs. This can be explained based on the fact that *Move Method* does not affect as many entities as other refactorings, thus its impact is limited.

It is important to point out here that even after applying proper refactorings, the tracing performance, especially in terms of precision, is still far from being optimal. This suggests that some work is needed on the other side of the trace link i.e. the requirements themselves. Just like source code, requirements can get outdated, and in order to keep the system quality under control, they have to evolve too [51]. The approach presented in this paper can be used to support such a process. This can be achieved by observing hard-to-trace requirements. These requirements that result in low precision and recall, even after refactoring the code, represent candidates for updating. In other words, our approach can be used to guide the requirements evolution process [52], [53].

VII. RELATED WORK

In this paper, we addressed some of the issues associated with code evolution that might affect the performance of traceability methods. Related to our work is the work in [54], in which Antoniol *et al.* developed a tool to establish and maintain traceability links between subsequent releases of an Object-Oriented software system. The method was used to analyze multiple releases of two projects. Using such information, the tool recovers design from the code, compares recovered design with the actual design and helps the user to deal with inconsistencies during evolution. Later in 2004, Antoniol *et al.* [55] 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. Cases such as class replacement, class merge and split were considered in their analysis. Our approach is distinguished from this work in the sense that we use refactoring as a preprocessing step to enhance the performance, rather than dealing with the implications of already applied refactorings.

Hammad *et al.* [56], proposed an approach to maintaining code-to-design traceability during code evolution. In particular, they developed a tool that uses lightweight analysis and syntactic differencing of the source code changes to detect when a particular code change may have broken traceability links. Design documents are updated accordingly. Evaluation of their approach showed that their approach outperforms a manual inspection approach. Grechanik *et al.* [48] proposed an approach for automating parts of the process of recovering traceability links between types and variables in java programs and elements of the system's use case diagrams. The proposed approach searches for patterns in the names and the values of program entities, and uses such patterns to recover traceability links between these entities and elements of the system's use cases in evolving software systems. Our work supports such approach by restoring identifiers names, thus recovering missing patterns in the system.

The approach proposed by Charrada *et al.* [57] tackles the problem we tackle in this paper from a different perspective. In particular, they proposed an approach to automatically identifying outdated requirements. This approach is based on 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 lexical structure from the code side, their approach can work on the same problem but from the opposite side of the trace link (the trace query). This will accelerate the process of bridging the textual gap in the system.

Finally, since this paper starts with Gotel and Finkelstein's definition of traceability [1], and is based on Gotel and Morris's theoretical approach of IR-based automated tracing [13], 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.* identified a number of challenges for implementing effective software and systems traceability [58]. In the set of short-term goals 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. 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.

VIII. Conclusions

In this paper, we proposed a novel approach for enhancing the performance of automated tracing tools using refactoring. In particular, we described an experiment for assessing the effect of applying various types of refactorings on the different performance aspects of IR-based tracing methods. Our main hypothesis is that certain refactorings will re-establish the decaying lexical structure 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 categories of 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 categories of refactorings to mitigate these problems. Results showed that restoring textual information in the system (e.g. *Rename Identifier*) has a significantly positive impact on the performance in terms of recall and precision. In contrast, refactorings that remove redundant information (e.g. *Extract Method*) affect the performance negatively. The results also showed that moving information between the system's modules (e.g. *Move Method*), has no noticeable impact on the performance.

Our results address several issues related to code and requirements evolution, as well as applying refactoring as a performance enhancement strategy. Future work in this domain includes evaluating other types of refactoring (Table I) over industrial length datasets. In addition, other issues related to code evolution, especially changes in the code structure (e.g., remove class, add class, change inheritance relations, etc.) will be investigated.

Acknowledgment

We would like to thank the partner company for the generous support of our research. This work is supported in part by the U.S. NSF (National Science Foundation) Grant CCF1238336.

References

[1] O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in *Intrenational Conference on Requirements Engineering*, 1994, pp. 94–101.

[2] A. De Lucia, R. Oliveto, F. Zurolo, and M. Di Penta, "Improving comprehensibility of source code via traceability information: A controlled experiment," in *International Conference on Program Comprehension*, 2006, pp. 317–326.

[3] J. H. Hayes, A. Dekhtyar, and S. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[4] A. von Knethen, B. Paech, F. Kiedaisch, and F. Houdek, "Systematic requirements recycling through abstraction and traceability," in *International Conference on Requirements Engineering*, 2002, pp. 273–281.

[5] A. von Knethen, "Automatic change support based on a trace model," in *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2002.

[6] D. Poshyvanyk, Y. gal Guhneuc, and A. Marcus, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[7] G. Spanoudakis and A. Zisman, "Software traceability: A roadmap," *Handbook of Software Engineering and Knowledge Engineering*, vol. 3, pp. 395–428, 2004.

[8] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards, "Lessons learned from implementing requirements traceability," *Crosstalk – Journal of Defense Software Engineering*, vol. 8, no. 4, pp. 11–15, 1995.

[9] J. Cleland-Huang, R. Settimi, and E. Romanova, "Best practices for automated traceability," *Computer*, vol. 40, no. 6, pp. 27–35, 2007.

[10] A. De Lucia, R. Oliveto, and G. Tortora, "Assessing ir-based traceability recovery tools through controlled experiments," *Empirical Software Engineering*, vol. 14, no. 1, pp. 57–92, 2009.

[11] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *International Conference on Software Engineering*, 2003, pp. 125–135.

[12] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, pp. 970–983, 2002.

[13] O. Gotel and S. Morris, "Out of the labyrinth: Leveraging other disciplines for requirements traceability," in *IEEE International Requirements Engineering Conference*, 2011, pp. 121–130.

[14] M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *Journal of System and Software*, vol. 1, no. 3, pp. 213–221, 1984.

[15] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in *International Conference on Program Comprehension*, 2009, pp. 188–197.

[16] N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system," in *Conference of the Centre for Advanced Studies on Collaborative Research*, 1998, pp. 4–14.

[17] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: a silver bullet for traceability recovery," in *International Conference on Software Maintenance*, 2006, pp. 299–309.

[18] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," in *International Conference on Automated Software Engineering*, 2010, pp. 245–254.

[19] W. Opdyke, *Refactoring Object-Oriented Frameworks*. Doctoral thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[20] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1–12, 1998.

[21] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwé, "On the use of clone detection for identifying crosscutting concern coden," *IEEE Transactions on Software Engineering*, vol. 31, pp. 804–818, 2005.

[22] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[23] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.

[24] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *European Conference on Software Maintenance and Reengineering*, 2003, pp. 91–100.

[25] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving usability of software refactoring tools," in *Australian Software Engineering Conference*, 2007, pp. 307–318.

[26] M. Katić and K. Fertalj, "Towards an appropriate software refactoring tool support," in *WSEAS International Conference on Applied Computer Science*, 2009, pp. 140–145.

[27] N. Anquetil, C. Fourrier, and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *Working Conference on Reverse Engineering*, 1999, pp. 235–255.

[28] F. Fontanaa, P. Braionea, and M. Zanonia, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 1–8, 2011.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[30] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[31] "Bad smell identification for software refactoring," [Online] www.jdeodorant.org, 2013.

[32] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.

[33] J. Mayrand, C. Leblanc, and E. M. Merlo, "Refactoring support based on code clone analysis," in *International Conference on Software Maintenance*, 1996, pp. 244–253.

[34] E. DualaEkoko and M. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Transactions on Software Engineering Methodology*, vol. 20, no. 1, pp. 1–31, 2010.

[35] D. Wilking, U. Kahn, and S. Kowalewski, "An empirical evaluation of refactoring," *e-Informatica Software Engineering Journal*, vol. 1, no. 1, pp. 44–60, 2007.

[36] M. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in *International Symposium on Empirical Software Engineering*, 2006, pp. 297–306.

[37] "Duplicated code detection tool (sdd)," [Online] wiki.eclipse.org/Duplicated_code_detection_tool_(SDD), 2013.

[38] A. Meneely, B. Smith, and L. Williams, *iTrust Electronic Health Care System: A Case Study*. Springer, 2012, ch. Software and Systems Traceability.

[39] F. Porter, "Readings in information retrieval." Morgan Kaufmann Publishers Inc., 1997, ch. An algorithm for suffix stripping, pp. 313–316.

[40] A. Mahmoud and N. Niu, "Source code indexing for automated tracing," in *International Workshop on Traceability in Emerging Forms of Software Engineering*, 2011, pp. 3–9.

[41] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *International Conference on Program Comprehension*, 2012, pp. 183–192.

[42] S. Sundaram, J. H. Hayes, A. Dekhtyar, and E. Holbrook, "Assessing traceability of software engineering artifacts," *Requirements Engineering Journal*, vol. 15, no. 3, pp. 313–335, 2010.

[43] C. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[44] J. Aslam, E. Yilmaz, and V. Pavlu, "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*, 2005, pp. 573–574.

[45] H. Sultanov, J. H. Hayes, and W.-K. Kong, "Application of swarm techniques to requirements tracing," *Requirments Engineering Journal*, vol. 16, no. 3, pp. 209–226, 2011.

[46] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[47] A. Dean and D. Voss, *Design and Analysis of Experiments*. Springer, 1999.

[48] M. Grechanik, K. McKinley, and D. E. Perry, "Recovering and using usecasediagramtosourcecode traceability links," in *joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 95–104.

[49] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *European Conference on Software Maintenance and Reengineering*, 2010, pp. 81–90.

[50] K. S. Jones, "utomatic summarising: The state of the art," *Information Process Management*, vol. 43, no. 6, pp. 1449–1481, 2007.

[51] T. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Software*, vol. 20, no. 6, pp. 35–39, 2003.

[52] A. Aurum and C. Wohlin, "Requirements experience in practice: Studies of six companies," in *Engineering and Managing Software Requirements*, P. A. Laplante, Ed., 2005, pp. 405–426.

[53] E. Ben Charrada, A. Koziolek, and M. Glinz, "An automated hint generation approach for supporting the evolution of requirements specifications," in *Joint ERCIM Workshop Software Evolution (EVOL) and Int. Workshop Principles of Software Evolution*, 2010, pp. 58–62.

[54] G. Antoniol, A. Potrich, P. Tonella, and R. Fiutem, "Evolving object oriented design to improve code traceability," in *International Workshop on Program Comprehension*, 1999, pp. 151–160.

[55] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *International Workshop on Principles of Software Evolution*, 2004, pp. 31–40.

[56] M. Hammad, M. Collard, and J. Maletic, "Automatically identifying changes that impact code-to-design traceability during evolution," *Software Quality Control*, vol. 19, no. 1, pp. 35–64, 2011.

[57] E. Ben Charrada, A. Koziolek, and M. Glinz, "Identifying outdated requirements based on source code changes," in *International Requirements Engineering Conference*, 2012, pp. 61 –70.

[58] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, and G. Antoniol, "The quest for ubiquity: A roadmap for software and systems traceability research," in *International Conference on Requirements Engineering*, 2012, pp. 71–80.