Traceability-Enabled Refactoring for Managing Just-In-Time Requirements

Nan Niu[*], Tanmay Bhowmik[†], Hui Liu[‡], and Zhendong Niu[‡]
[*] Department of Electrical Engineering and Computing Systems, University of Cincinnati, USA
[†] Department of Computer Science and Engineering, Mississippi State University, USA
[‡] School of Computer Science and Technology, Beijing Institute of Technology, China
nan.niu@uc.edu, tb394@msstate.edu, {liuhui08, zniu}@bit.edu.cn

***Abstract*—Just-in-time requirements management, characterized by lightweight representation and continuous refinement of requirements, fits many iterative and incremental development projects. Being lightweight and flexible, however, can cause wasteful and procrastinated implementation, leaving certain stakeholder goals not satisfied. This paper proposes traceability-enabled refactoring aimed at fulfilling more requirements fully. We make a novel use of requirements traceability to accurately locate where the software should be refactored, and develop a new scheme to precisely determine what refactorings should be applied to the identified places. Our approach is evaluated through an industrial study. The results show that our approach recommends refactorings more appropriately than a contemporary recommender.**

***Index Terms*—requirements management; just-in-time requirements; traceability; refactoring;**

I. Introduction

For many successful and long-lived software projects, the traditional notions of up-front requirements engineering (RE) — where requirements are formally specified and then baselined for subsequent development — offer limited value [1, 2]. Instead, requirements in these projects are captured less formally and become more elaborated only after the implementation begins. This is known as *just-in-time* RE [3, 4] which encompasses two key aspects: lightweight representation (defining requirements at a convenient level of detail) and evolutionary refinement (clarifying them when needed). Projects adopting such practice include both open source systems like Mozilla [4] and proprietary ones such as IBM's Rational Team Concert [5].

Managing requirements in a just-in-time manner is not without risk. One challenge refers to the stagnation of the requirements that developers have already started to implement. A recent study investigated 157 user stories and showed that about 10% failed to be realized even after certain coding efforts were made [5]. Another major issue concerns the hindrance to the fulfillment of future requirements. Just-in-time RE, as currently practiced, lacks explicit "big picture" thinking [4]. As a result, a long-term penalty is often imposed on how well the software can be extended to accommodate new and changing requirements.

A mainstream method to facilitate future extensions and adaptations of a software system is refactoring [6, 7]. The basic idea of refactoring is to keep the internal software quality under control during evolution by applying a set of behavior-preserving transformations to the code base. It is our overarching hypothesis that these transformations, if applied appropriately, can advance the state of the practice in software engineering by enabling more just-in-time requirements to be fully satisfied.

Testing this hypothesis entails addressing several specific research questions, including where, when, and how frequently to refactor, what refactoring(s) to perform, and how to assess the refactoring effect [8]. Murphy-Hill and Black [9] refer to the "when" and "how frequently" choices as refactoring tactics, and further suggest "refactoring early and refactoring often" is the principle that will benefit most developers. Contemporary approaches, however, depart from this principle by relying passively on developer's spontaneity to trigger refactoring. Consequently, a number of refactorings may be missed or conducted later than expected [10]. The impacts are negative: Few refactorings can result in poor software quality and delayed refactorings can incur high maintenance cost.

In our earlier work, we devised an instant refactoring framework for actively assisting developers with better tactics [11]. The framework deploys a monitoring mechanism to analyze code changes constantly (e.g., between saves), and then prompts the developer for potential refactorings. The direct effect is that more refactorings are made without much delay [11]. We thus believe this framework sets up a firm and promising basis for handling those just-in-time requirements that are slowly progressing or hard to accommodate. While our work so far has focused on the "when" and "how frequently" tactics [11, 12], identifying "where" to apply "what" refactoring(s) remains one of the most pressing and vital issues confronting the research community [8].

In this paper, we tackle this pivotal issue via the novel use of requirements traceability. To locate more accurately "where" the code base should be refactored, we synthesize clustering-based link retrieval [13] with the as-needed traceability information captured in issue tracking repositories [4]. To determine more precisely "what" refactoring(s) should be applied to the identified places, we develop a new scheme by first examining requirements semantics as they relate to implementation, then leveraging the semantic characterization to uncover the problems in the code (i.e., bad smells [7]) that may impede the fulfillment of the requirements, and finally choosing the type of refactorings that can remove the bad smells.

The contributions of our work lie in the strategic use of requirements traceability to change how code refactoring can be practiced to better fit its purpose — to make the software more extensible and adaptable to new requirements [8, 9]. In our opinion, the work represents an example of innovation *through* RE[1] in that an extensively researched requirements topic, namely traceability [14], creates principled ways to improve system development. In what follows, we present background information on just-in-time RE and software refactoring in Section II. We then detail our traceability-enabled refactoring framework in Section III. Section IV describes an industrial study to evaluate our approach. Finally, Section V places our work in the context of related research and Section VI concludes the paper.

II. Background

A. *Just-In-Time Requirements Analysis*

In a recent attempt to understand RE in open source software projects, Alspaugh and Scacchi [2] identified three characteristics of what they termed "classical requirements": 1) a central document (e.g., an IEEE-830 software requirements specification); 2) described in terms of problem space constructs (e.g., stakeholder goals); and 3) examination of the document to uncover defects (e.g., ambiguity, inconsistency, and incompleteness). This last point offers probably the greatest benefit, as asserted by Boehm with data support [15]:

> Clearly, it pays off to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.

Because of this benefit, the effort spent in getting the requirements right (e.g., modeling and formal methods) early in the software life cycle can be justified. Ernst and Murphy [4] referred it as "up-front RE".

What was found out, however, is ongoing development with neither classical requirements [2] nor up-front RE [4]. This is true for a number of long-lived and critical open source systems that provide quality and rich functional capabilities, such as Apache and Mozilla [16].

What was actually practiced, then, is regarded as "just-in-time RE" where requirements are expressed less formally and only fully elaborated once the implementation begins [3, 4]. Contrary to up-front RE's detailed plans and estimates for what should be done, just-in-time RE promotes rapid development and continuous refinement. The basic tenet is given succinctly by Beck [17]:

> As we develop, we learn more about the problem and what is required to solve the problem.

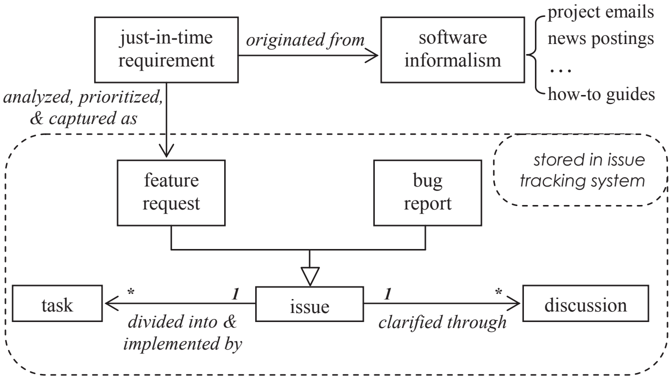


Fig. 1. Just-in-time requirements management via issue tracking system.

The common and dominant way to manage just-in-time requirements is the use of issue tracking system, e.g., Bugzilla[2] or JIRA[3]. We model the relevant concepts in Fig. 1 by synthesizing related studies in the literature [2, 4, 5, 16, 18]. The model is by no means a complete reference but is intended to serve as a unified conceptual basis for streamlining our discussion. Scacchi [16] recognized in his seminal work a set of "software informalisms" as the main sources for requirements. Informalisms, like emails and instant messaging, provide socially lightweight mechanisms for communicating and coordinating project knowledge.

Just-in-time requirements may be represented by user stories or in a feature list [4], but as shown in Fig. 1, the to-be-implemented ones "eventually end up as feature requests in an issue tracking system" [18]. Feature request and bug report are two special kinds of issue. For our discussion, they are different because "a defect (bug) stops living once resolved, but the description of a requirement (feature) is still valid documentation once implemented" [18]. The implementation of a feature typically involves a series of clarifying discussions. The actual implementation is then carried out by dividing the feature request into concrete and actionable tasks and by assigning the tasks to developers.

Because of the rich information stored in issue tracking systems, they are adopted by many projects besides open source systems to keep the requirements on track. For example, IBM's Rational Team Concert project, which follows an iterative development process with six-week release cycles, relies on an issue tracking system for geographically distributed sites to clarify the requirements and to coordinate developers' implementation tasks [5]. Studying the requirements clarification patterns in this project reveals a mixed outcome of just-in-time RE. While certain requirements progress smoothly toward full implementation, others are clarified late in the development cycle which makes them infeasible to be realized [5]. Two of the suspicious patterns are displayed in Fig. 2. They are representative of just-in-time RE's drawbacks. Because requirements are defined at a convenient level of detail, erroneous and wasteful implementation can happen, as shown

[1] RE'14 has a dual theme focused on innovation: innovation *in* RE and innovation *through* RE. See http://re14.org for more information.

[2] http://www.bugzilla.org

[3] https://www.atlassian.com/software/jira

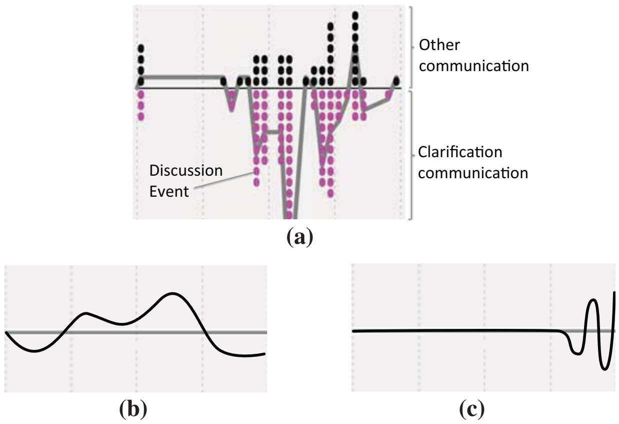


Fig. 2. Requirements clarification patterns where time in each graph proceeds left to right and depicts a release cycle. **(a)** Transforming discrete discussions (dots) into a pattern trajectory (line). **(b)** Back-to-draft. **(c)** Procrastination. All the graphs, **(a)**–**(c)**, are adopted from [5].

by the "back-to-draft" pattern in Fig. 2b. Because requirements are clarified when needed, developers sometimes wait till the last minute to engineer them, as shown by "procrastination" in Fig. 2c. The procrastinated implementation is at greater risk of not being completed, but even if it is finished before the release, higher maintenance cost may be incurred [4].

In sum, being lightweight and flexible makes just-in-time RE a better fit for many software development projects than classical up-front RE, but often this comes at the cost of not fully meeting the project's and stakeholders' goals.

B. Software Refactoring

Although developers have long practiced program restructuring, the term "refactoring" was first introduced by Opdyke [6] as "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure". The core idea is to redistribute programming elements (e.g., variables and methods) across the class hierarchy in order to facilitate future adaptations and extensions [8]. Clearly, the main objective of refactoring is to help adequately prepare the software system for meeting new and changing *requirements*.

Fowler [7] popularized refactoring when he cataloged 72 structural changes observed repeatedly in various programming languages and application domains. Research confirms that refactoring has become mainstream and is now widely practiced by developers [9, 10, 19]. A state-of-the-art survey by Mens and Tourwé [8] shows that the refactoring process consists of a number of distinct activities:

1. Identify where the software should be refactored.
2. Determine what refactoring(s) should be applied to the identified places.
3. Ensure that the applied refactoring preserves behavior.
4. Apply the refactoring.
5. Assess the effect of the refactoring.
6. Maintain the consistency between the refactored code and other artifacts such as requirements and tests.

Performing these activities requires different tools as the manual approach is often repetitive, time-consuming, and error-prone — an example would be to apply Rename Identifier refactoring across multiple class files. One area in which extensive tool support has been developed is "bad smell" detection. Bad smells are "structures in the code that suggest the possibility of refactoring" and 22 such signs of potential problems are initially identified in [7].[4] A review of these 22 code smells suggests an imbalanced support in that **Duplicated Code** receives most research attention whereas some smells like **Message Chains** receive little [20]. The review also points out the lack of evidence available to justify the use of current smell-based approaches to direct effective refactoring [20]. Thus our knowledge about "where" to apply "what" refactoring(s) remains insufficient.

The list of activities given by Mens and Tourwé [8] is especially important from the viewpoint of developing automated tool support. Human intervention, however, is indispensable in software refactoring. A primary reason stems from the intrinsically informal interpretation of "behavior preservation". Although Opdyke's original definition hinges on formally checking refactoring preconditions and input-output behaviors [6], it is later proven to be insufficient [8]. Rather, it is recommended that we should have a wider range of definitions of behavior that need to be preserved by refactoring, depending on domain-specific or even user-specific concerns [8]. For example, the temporal constraints should be preserved when real-time software is refactored, and so should the concrete notions of safety (e.g., liveness) be preserved in the refactoring of mission-critical systems.

Another area that automated support is limited is the tactical dimension of refactoring choices. Such decisions include not only the previously mentioned "when" and "how frequently" to refactor [9], but also how to mix refactoring with developers' other programming tasks (e.g., bug fixing). On one hand, refactoring can be conducted over protracted periods, during which developers perform few if any other kinds of program changes. For example, Microsoft typically reserves 20% of development efforts on thorough refactoring right after a version is released and before the development of the next version is initiated [21]. On the other hand, refactoring can be seen as a means to reach a specific end (e.g., adding a feature) instead of the end itself. For example, more than half of the Mylyn refactorings indicate that developers intersperse other kinds of program changes with refactorings to keep the code healthy [10]. These competing but often coexisting tactics make it difficult to tease out and measure the impact of refactoring.

In sum, refactoring is a mainstream software engineering practice whose prime purpose is to make the code base more extensible and easily adapted to future requirements. Despite the rapidly growing body of research in the last decade, there remains an important gap in the current knowledge about how to locate the "where" and determine the "what" for effective refactoring. The emerging research thrust in refactoring tactics

[4] More bad smells, together with refactorings that may help dispel them, can be found at http://refactoring.com .

further illuminates the challenge of assessing the value and effect of refactoring.

III. Enabling Software Refactoring Through Requirements Traceability

The underlying premise of our research is that using just-in-time requirements to drive software refactoring will advance the state-of-the-art in both fields. Our essential argument is that the role of refactoring is better justified by leading more requirements to their full implementation. To examine this premise, we significantly extend monitor-based refactoring [11] by exploiting the novel use of requirements traceability information.

An overview of the proposed framework is presented in Fig. 3. As the developer performs coding and debugging activities, our framework monitors the code changes and relates the programming activities to requirements with the possible help of the issue tracking system. Through tracing and semantic characterization of the requirements, our framework recommends refactoring opportunities to the developer.

Compared to our prior work [11], the key difference which is also the unique advantage of the approach in Fig. 3 is *requirements-driven*. Unlike contemporary refactoring approaches that scan largely the source code to detect bad smells, our framework makes full and explicit use of the requirements information (i.e., feature requests extracted from the issue tracking system) to direct the refactoring process. In this way, refactoring is made to not just clean up the unhealthy code, but to do so with a purpose — to be better ready for satisfying more requirements. The rest of this section describes in detail how we leverage requirements traceability (Section III-A) and a new semantic scheme (Section III-B) to drive software refactoring. Section IV presents an evaluation of our approach.

A. Using Traceability to Locate "Where"

The specific question we address in this sub-section concerns the accurate identification of the places in the code that should be refactored. In our framework, we first consider the set of requirements that the current development cycle implements. This set can be readily extracted according to the issue tracking system's "feature requests" (cf. Fig. 1), which we call "targeted requirements" with respect to the ongoing implementation. We then retrieve requirements-to-source-code traceability links for the targeted requirements, and detect code smells[5] only within the tracing results. The simple rule that we are using here is that: "Do not refactor the code that cannot even be traced to the targeted requirements". Thus the question of locating to-be-refactored code is cast as a requirements tracing problem.

Traceability is one of the most actively and intensively researched topics in RE [22, 23, 24, 25, 26]. A tracing method is *accurate* if it achieves a high level of recall and precision, where recall is defined as the percentage of correct links that are retrieved and precision is defined as the percentage of retrieved links that are correct [27]. It is these accuracy criteria that our framework uses to guide the identification of "where" the software should be refactored. Although numerous tracing methods exist, the ones adopting standard information retrieval (IR) algorithms are equivalent in that they are able to capture almost the same traceability information [28]. In most cases, a recall of 90% is achievable at precision levels of 5-30% [24, 27, 28, 29].

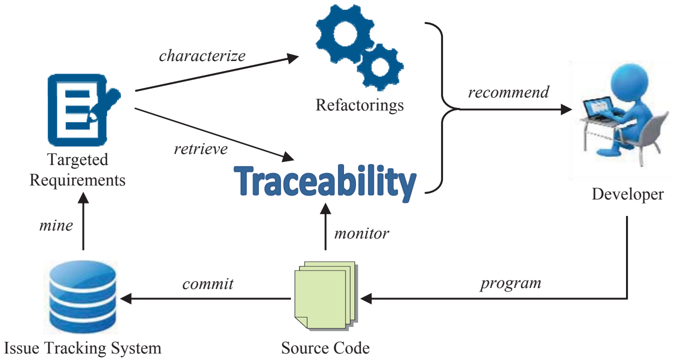


Fig. 3. Traceability-enabled refactoring framework.

Before choosing an accurate tracing mechanism among the many options [14, 30] or even developing a new one, we must understand how traceability is currently practiced in just-in-time RE. The pertinent literature, however, is surprisingly scant. A noticeable exception is Ernst and Murphy's recent empirical study on three large open source software projects [4]. The tracking from requirements to implementation in these projects, perhaps not surprisingly, is labeled "as-needed traceability" [4]. Inheriting the essential characteristics of just-in-time RE, traceability is implemented where necessary, but as long as it *is* implemented, traceability is defined through the built-in support from the issue tracking system. All three projects follow the practice of prefacing a commit message to the code repository with a task ID number. In another word, the requirements-to-source-code traceability links can be obtained by querying the issue tracking system through the feature request's task ID(s). The advantage of as-needed traceability is that developers actually do it [4]. It is therefore emphasized that automated traceability recovery is fruitless without taking into account the as-needed traceability information [4].

It should be pointed out that as-needed traceability does not equal to perfect traceability. For the requirements whose implementation is not yet finished, that is, those feature requests whose status is still "open", the traceability information stored in the issue tracking system is incomplete at best. Even for the "closed" ones, if their implementation is procrastinated as illustrated in Fig. 2c, then the traceability information may be completely missing in the issue tracker due to the time pressure. Thus the requirements-to-source-code links defined by as-needed traceability are of high precision, i.e., they are indeed correct but are hardly complete.

Our framework integrates as-needed traceability into clustering-based link retrieval — a technique that we investigated in [13]. The rationale is derived directly from the

[5] The smell detection and refactoring recommendation in our framework are also requirements-driven and will be discussed in Section III-B.

TABLE I
Guiding smell detection and refactoring selection via requirements' semantic analysis

Requirement's Action Theme		Issue_ID*	Code Smell	Refactoring
Add	functional capability	226680	**Switch Statements**	Introduce Local Extension
	stakeholder role	839959	**Large Class** **Duplicated Code**	Extract Class
Enhance	quality attribute	578133	**Long Method**	Substitute Algorithm
Refine	expansion (more details)	228585	**Message Chains**	Add Parameter
	extension (more specifics)	713036	**Long Parameter List**	Extract Subclass
Manage Dependency	abstraction	341665	**Duplicated Code**	Extract Superclass Pull Up Method
	invocation	239532	**Long Method** **Switch Statements**	Replace Method with Method Object
Remove	redundant feature	229482	**Duplicated Code** **Feature Envy**	Remove Code Move Method
Adapt	another operating system	251722	**Parallel Inheritance Hierarchies**	Extract Interface
	another platform	669642	**Primitive Obsession**	Dynamic Method Definition

[*] All the feature requests can be accessed from Mozilla's issue tracker at https://bugzilla.mozilla.org/show_bug.cgi?id=Issue_ID

```
1.  For each req $\in$ set of targeted requirements
2.      VSM $\leftarrow$ retrieve candidate traceability links based on the
            tf-idf textual similarity between code and req
3.      C $\leftarrow$ cluster VSM by applying single-link with $k$=8
4.      $C_{req}$ $\leftarrow$ remove 3 lowest-quality clusters from C
5.          If as-needed traceability $\neq$ NULL
6.              $C_{req}$ $\leftarrow$ remove those clusters containing no
                    as-needed traceability from $C_{req}$
7.      Return $C_{req}$
```

Fig. 4. Identifying to-be-refactored code by integrating as-needed traceability into clustering-based link retrieval.

cluster hypothesis stating that correct links tend to cluster near other correct links and farther away from incorrect ones. We examined five clustering algorithms (k-means, bisecting divisive, single-link, complete-link, and average-link) over four datasets, and the findings showed that the single-link algorithm at the 8-cluster (k=8) granularity best supports the cluster hypothesis in automated traceability [13]. The quality of the resulting clusters can then be judged on the basis of the link that is most similar to the requirement, and discarding the 3 lowest-quality clusters is found to significantly improve the tracing accuracy [13]. While this heuristic is kept intact, we introduce an additional filter so that if the as-needed (and correct) traceability links are defined, then the clusters that do not contain these links are further removed. The entire procedure is summarized in Fig. 4.

Note that clustering is only one way of feeding as-needed traceability into the retrieval process. Classification, for example, could lead to another automated solution [31]. Nevertheless, our underlying mechanism (lines 2–4 of Fig. 4) is shown to greatly enhance a baseline IR method [13]. The augmented filter (lines 5–6 of Fig. 4), as will be shown in Section IV, further improves tracing accuracy.

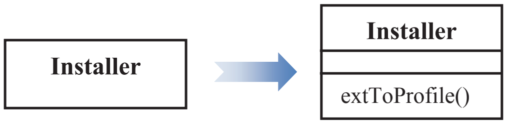
B. Characterizing Semantics to Determine "What"

Having identified the to-be-refactored code through traceability, we now turn our attention to refactoring selection. Current research chooses refactorings depending mainly on the bad smells directly observed in the source code. However, bad smells are symptoms of design problems [7]. Different problems can lead to the same symptom, but their resolutions may differ. To handle a smell of **Duplicated Code**, for instance, we can simply Remove Code to reduce redundancy or Extract Class to achieve a better separation of concerns. A more extreme example is provided in Fowler's catalog: if a class is affected by **Type Code**, one can replace it in 5 different ways: Class, Module Extension, Polymorphism, State/Strategy, and Subclasses [7]. Advancing the research on refactoring selection therefore requires a high *precision*, i.e., recommending the exact type of refactoring(s) by looking beyond the symptom (code smell).

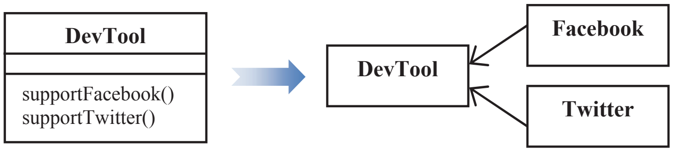
Our framework shown in Fig. 3 employs a *requirements-driven* component to direct smell detection and refactoring selection. In particular, we build upon the work on semantic analysis in RE [32, 33] to tease out the action-oriented theme of a requirement. Such theme, according to Fillmore's case theory [34], can be characterized by the *verb* in the requirements description and the *direct object* that the verb evokes (acts upon). We then detect those code smells that pose potential threats to the implementation of the intended action, and further recommend refactoring(s) to mitigate the threats. Table I shows the novel scheme, in which the requirements semantics helps unearth *why* the code smells so as to precisely determine *how* to correct the smell.

To guide the semantic classification, we extend the previous research [32, 33] and manually inspect the feature requests of Mozilla — a project that adopts just-in-time RE [4]. The analysis of Mozilla's issue tracking system was carried by two researchers independently. Their findings were then shared and discussed with the participation of a third researcher. Collectively, the most representative requirement was selected to report here. Next we ground our discussion on each of the semantic categories by using these representative requirements. Notably, the just-in-time requirements are stated heavily in solution-space terms, a deviation from "classical requirements" [2]. This, however, effectively facilitates our mapping between the requirement and its implementation.

1.1) Add a functional capability: Feature request 226680 is about providing a checkbox for installing extensions to profile directory so that the installation can be simplified. In complex procedures like installation, **Switch Statements** often appear but they violate object-orientation's encapsulation principle. To lead to the creation of a new functional capability, Introduce Local Extension can be applied. Such a transformation is depicted as follows.

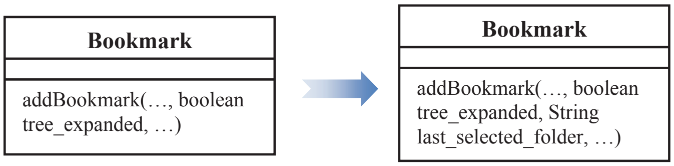


1.2) Add a stakeholder role: Issue 839959 requests new APIs (application programming interfaces) to be enabled for social media providers (e.g., Facebook, Twitter, etc.) to have direct access to Firefox's devtools. If **Large Class** with **Duplicated Code** exists in the code base, then one can Extract Class to better modularize the individual stakeholder(s). This refactoring is illustrated below.

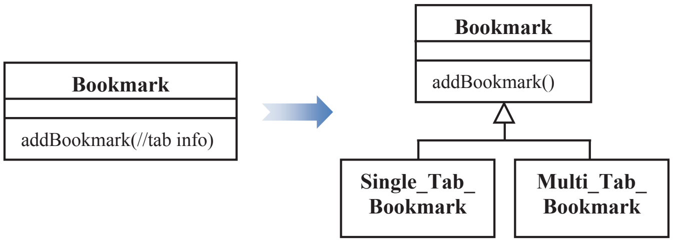


2) Enhance a quality attribute: For issue 578133, the heading reads, "make us fast". **Long Method** can cause not only performance issues but degradation of other quality attributes like complexity and understandability. Substitute Algorithm helps replace the smell with the code that is faster, simpler, clearer, etc. Note that "make us fast" is truly a just-in-time requirement because its meaning becomes more unambiguous only after the implementation begins [4].

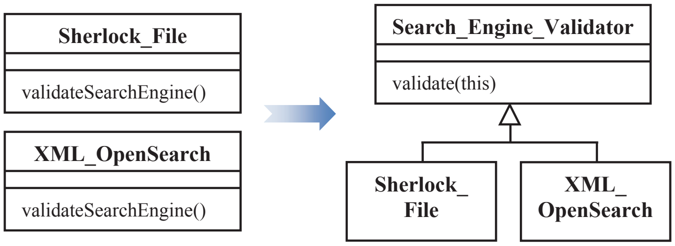
3.1) Refine by expanding the requirement with more details: Requirement 228585 builds on the bookmark-adding feature that remembers if the user last had the tree view expanded or not, and suggests a step further to remember the last selected folder in the tree view so that the folder can be highlighted. The intertwining relationship may cause **Message Chains** in the implementation, which can be addressed by Add Parameter as follows.



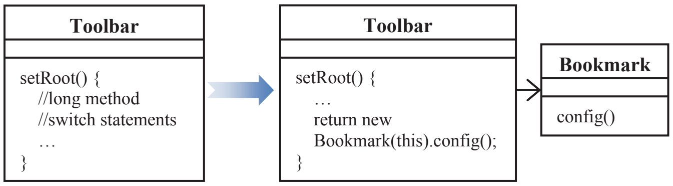
3.2) Refine by extending the requirement with more specifics: Feature request 713036 suggests extending bookmark operations from a single-tab to a multi-tab setting. **Long Parameter List** is a sign of potentially tangling code. The refactoring Extract Subclass delineated below can be used to refine the inheritance hierarchy.



4.1) Manage dependency via abstraction: Issue 341665 acknowledges the use of Sherlock file to validate search engine, but points out that there is no technical reason XML OpenSearch description could not be used. Validating search engine in different ways can result in scattered and **Duplicated Code**. Extract Superclass, along with Pull Up Method, can then be performed to improve the code structure. This transformation is sketched as follows.



4.2) Manage dependency via invocation: Issue 239532 is raised because the user wants to allow the root of toolbar to be set with the bookmark folder. The tight interdependency between setting toolbar's root and various browser widgets can cause **Long Method** and **Switch Statements**. As shown below, Replace Method with Method Object helps overcome the potential problems of these smells.



5) Remove a redundant feature: Issue 229482 requests to remove the status bar of bookmark as the displayed information is made available already in the manager window. Two code smells are relevant here. If displayStatusBar() of the Bookmark class is **Duplicated Code**, then Remove Code can be triggered. If the method is too interested in another class (Manager_Window), it is a sign of **Feature Envy** and Move Method over Manager_Window will achieve a higher-cohesion and lower-coupling redistribution.

6.1) Adapt to another operating system (OS): Issue 251722 wants Mozilla running on Linux and Mac to offer the option to import saved passwords same as Windows. Supporting multiple OSs can cause **Parallel Inheritance Hierarchies**, which may be refactored with EXTRACT INTERFACE.

6.2) Adapt to another platform: Feature request 669642 points out that about:firefox pops up the description in desktops, but in mobile platform, about:firefox should be better implemented by navigating to a window where one can check for updates and find help information. Accordingly, DYNAMIC METHOD DEFINITION may address the problem of **Primitive Obsession** (e.g., literals used to express the about:firefox message).

It is worth mentioning here that the categories listed in Table I are classified manually, though future automation is possible, e.g., by calibrating the Naive Bayesian classifier [5, 31]. Our analysis of Mozilla's repository focuses on representative feature requests, though (semi-)automatically calculating the issue distribution over different categories can offer additional insight. Given the above reasons, the records presented in Table I may not be complete. In fact, at this stage of our research, we trade completeness for originality as our goal is to discover a new way to improve the precision of refactoring selection.

IV. EVALUATION

This section presents an empirical investigation into the impact of our traceability-enabled refactoring approach on just-in-time RE. While our overarching goal is to best transform and prepare the code to fulfill the requirements, the framework described in Section III is intended to achieve two specific objectives: (1) accurately locate the to-be-refactored code, and (2) precisely select the type of refactorings at these locations. Thus, we first introduce the background of our empirical study (Section IV-A), then assess separately how the two objectives are accomplished (Section IV-B and Section IV-C), and finally discuss the threats to validity of our study (Section IV-D).

A. Background

The subject system of our study is a 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 over a decade and is developed using Java and Eclipse technologies. It intends to meet the goals of multiple stakeholders, including job seekers, employers, educational institutions, and government agencies.

WDS development follows an iterative and incremental life cycle with each version released in around 6 to 8 weeks. While WDS developers are encouraged to "refactor early and refactor often", a designated refactoring period is reserved shortly after the latest version is released and right before the development of the next version is launched — a practice similar to Microsoft [21]. The requirements of WDS are managed in a typical just-in-time way. The project team uses the commercial state-of-the-practice JIRA issue tracking system to communicate the requirements and to record the as-needed traceability information.

We select the most recent stable release of WDS to investigate. We refer to this version as "code base X". This code base contains over 500 Java classes. The contextual information is shown in Fig. 5. The annotated t_a, t_b, and t_c in Fig. 5 should be thought of as short time periods (e.g., 1 or 2 days) rather than instantaneous time points. The just-in-time requirements extracted from JIRA for development cycle X include successfully closed feature requests {REQ_X$_1$, REQ_X$_2$, ..., REQ_X$_n$}, as well as three open features {REQ$_1$, REQ$_2$, REQ$_3$} that are not delivered at t_b. For the upcoming development cycle Y, two arriving requirements {REQ$_4$, REQ$_5$} are already entered into JIRA at t_b, although more features {REQ_Y$_1$, REQ_Y$_2$, ..., REQ_Y$_m$} may be requested in the future.

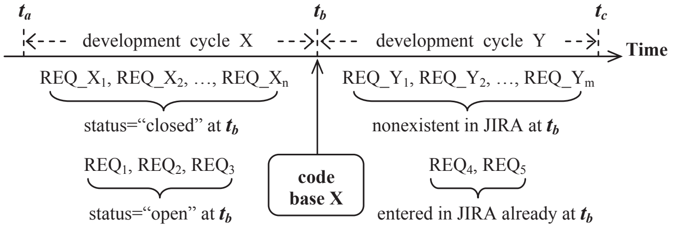
For code base X at t_b, the set of targeted requirements is composed of {REQ$_1$, REQ$_2$, REQ$_3$, REQ$_4$, REQ$_5$}. It is this set that our framework takes full advantage of (cf. Fig. 3). The middle column of Table II provides the requirements description. As-needed traceability for {REQ$_1$, REQ$_2$, REQ$_3$} can be found in JIRA, which shows partial implementation of these requirements. Further analysis of the discussion events indicates that REQ$_1$ and REQ$_2$ come "back-to-draft" (cf. Fig. 2b) whereas REQ$_3$'s implementation is procrastinated (cf. Fig 2c). For this reason, REQ$_1$ and REQ$_2$ are marked as infeasible during development cycle X.

B. Assessing Accuracy of Code Location

As we formulate refactoring location as a requirements tracing problem, the accuracy is measured by standard IR metrics: recall and precision [27]. The answer set of all the correct traceability links is provided by the WDS project team at the requirement-to-Java-class granularity. Table III compares the automated tracing results performed by the tf-idf textual matching with uniform pruning [27], the clustering-based retrieval [13], and the clustering-based retrieval augmented with the additional filter (cf. Fig. 4).

The results in Table III show that, compared to the term-based tf-idf retrieval, grouping candidate traceability links into clusters and then filtering out the 3 lowest-quality clusters improve both recall and precision for all the 5 targeted requirements. The main reason is that correct links are pulling other correct links toward each other but pushing incorrect links away to different clusters [13]. Note that the clustering-based link retrieval works regardless of the availability of as-needed traceability, indicating the generalizability of our approach beyond just-in-time RE.

For {REQ$_1$, REQ$_2$, REQ$_3$}, having as-needed traceability serve as an extra filter further enhances the tracing performance. While recall is unchanged from the clustering-based retrieval, precision increases because the filter discards more false positives — those links appearing in the cluster that does not include any of the correct as-needed traceability links. The improvement, when compared with the precision of REQ$_4$ and REQ$_5$, is nontrivial. Our results therefore stress



t_a
t_b
t_c
development cycle X
development cycle Y
Time
REQ_X$_1$, REQ_X$_2$, …, REQ_X$_n$
REQ_Y$_1$, REQ_Y$_2$, …, REQ_Y$_m$
status="closed" at t_b
nonexistent in JIRA at t_b
REQ$_1$, REQ$_2$, REQ$_3$
code base X
REQ$_4$, REQ$_5$
status="open" at t_b
entered in JIRA already at t_b

Fig. 5. Code base X (about 500 Java files) whose targeted requirements set at t_b is {REQ$_1$, REQ$_2$, REQ$_3$, REQ$_4$, REQ$_5$}.

TABLE II
DESCRIPTION AND CLASSIFICATION OF TARGETED REQUIREMENTS

ID	Description	Semantic Category (cf. Table I)
REQ$_1$	Provide a way to key youth achievements as part of student's academics	Refine by expansion (*3.1*)
REQ$_2$	Create the role of "WP Manager"† and allow it to run data integrity reports for individual staff	Add a stakeholder role (*1.2*)
REQ$_3$	Improve the merge utility to make it fast	Enhance a quality attribute (*2*)
REQ$_4$	Allow job-matching search on the public job order XML interface by accepting OSOC codes‡	Refine by expansion (*3.1*)
REQ$_5$	Extend search facility from a single company to self-service participants	Refine by extension (*3.2*)

† WP Manager (workforce program manager) is an emerging government role.
‡ OSOC (onetcenter.org) defines an occupation-specific national standard.

the importance of equipping automated traceability with the available as-needed information [4].

C. Assessing Precision of Refactoring Selection

We build on our tracing results to examine what refactoring(s) should be applied to the traced Java classes. Refactoring selection is essentially a recommendation problem [35]. In software engineering, a recommender helps to overcome developers' information overload by exposing them to the most interesting and relevant items — in our case, what code needs to be refactored in which way in order to correct what bad smells. Ultimately it is up to the developers to follow or ignore the recommendations.

Refactoring recommendations are often evaluated in two different ways. The first involves collecting opinions from domain experts, application developers, or other human evaluators. For example, Bavota *et al.* [36] recruited 30 Master's students in Computer Science to rate the usefulness of the suggested EXTRACT CLASS refactorings on a 5-point Likert scale. The second injects recommendations into actual development. For example, a prototype recommender, InsRefactor, was developed in our previous work and a controlled experiment was conducted to compare the students' refactoring performance with and without InsRefactor in order to quantify the recommendations' effectiveness [11].

We followed mainly the qualitative evaluation mechanism by asking the opinions from three WDS developers. In particular, we provided each developer with a hard copy of the targeted requirements (REQ$_1$–REQ$_5$) and asked them to individually rate the "precision" (1: very inappropriate, 2: not appropriate, 3: neutral, 4: somewhat appropriate, 5: appropriate) and the "perceived implementation difficulty" (1: very difficult, 2: somewhat difficult, 3: neutral, 4: easy, 5: very easy) of the recommended refactorings with respect to the targeted requirements.

TABLE III
ACCURACY OF LOCATING CODE FOR THE TARGETED REQUIREMENTS

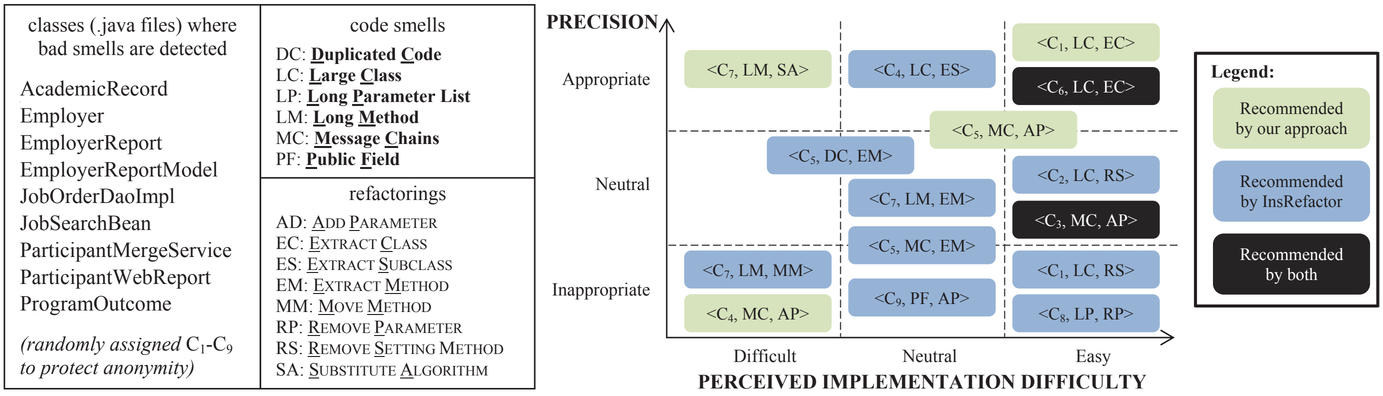
ID	**<Recall, Precision>**		
	term-based retrieval (tf-idf)	clustering-based retrieval	additional filter
REQ$_1$	<.91, .08>	<1.00, .20>	<1.00, .44>
REQ$_2$	<.73, .31>	<.95, .38>	<.95, .46>
REQ$_3$	<.80, .34>	<.85, .39>	<.85, .61>
REQ$_4$	<.82, .11>	<.82, .33>	<.82, .33>
REQ$_5$	<.85, .09>	<.88, .12>	<.88, .12>

The refactorings that the WDS developers assessed included the recommendations generated by our approach and the ones made by InsRefactor [11]. In our study, InsRefactor worked by taking the traced Java classes as input and calling a set of state-of-the-art smell detectors to recommend refactorings.[6] Although our approach used the same set of smell detectors, the refactorings were selected based on the scheme defined in Table I. This difference, though important for our internal data analysis, was *not* shown to the external evaluators (i.e., the WDS developers). In fact, all the recommendations, irrespective of the source, were displayed using the same style in Eclipse [11]. Basically the evaluator was prompted with the tuple <code, smell, refactoring> in an arbitrary order to provide the ratings, and optionally, the justifications.

Using *requirements* to drive refactoring is the unique feature of our approach. As shown in Table II, each of the 5 targeted requirements fits in one and only one of the semantic categories. This demonstrates the generalizability of our scheme, and more importantly, leads to refactorings more on the target. For the 9 classes listed in Fig. 6a, our approach makes 6 recommendations shown in Fig. 6b. InsRefactor, meanwhile, recommends 11 refactorings, two of which overlap with ours. Although InsRefactor's recommendations were perceived easier to implement (average=3.7; average of ours=3.5)[7], they were less appropriate (average=2.5; average of ours=3.8). Moreover, the refactorings targeted at previously infeasible requirements, namely REQ$_1$ and REQ$_2$, received positive ratings. One WDS developer commented, "Extract a WP Manager class is exactly what I'm doing right now [to implement REQ$_2$]." Not all of our recommendations received positive ratings. The one targeted at REQ$_3$ ("make the merge utility fast"), for instance, seemed appropriate but finding the actual algorithm to substitute for the long method was not an easy implementation task. The results thus show the precision of our refactoring selection and also indicate the room for improvement.

[6] A prototype implementation of InsRefactor as an Eclipse plug-in can be found at http://www.sei.pku.edu.cn/∼liuhui04/tools/InsRefactor.htm .

[7] Cohen's kappa of pairwise ratings among the three WDS developers is 0.77 for "precision" and 0.63 for "perceived implementation difficulty"; both show substantial inter-rater agreements [37].


classes (.java files) where bad smells are detected

AcademicRecord
Employer
EmployerReport
EmployerReportModel
JobOrderDaoImpl
JobSearchBean
ParticipantMergeService
ParticipantWebReport
ProgramOutcome

(randomly assigned C_1-C_9 *to protect anonymity)*

code smells

DC: **Duplicated Code**
LC: **Large Class**
LP: **Long Parameter List**
LM: **Long Method**
MC: **Message Chains**
PF: **Public Field**

refactorings

AD: Add Parameter
EC: Extract Class
ES: Extract Subclass
EM: Extract Method
MM: Move Method
RP: Remove Parameter
RS: Remove Setting Method
SA: Substitute Algorithm

(a) **(b)**

Fig. 6. **(a)** List of class files, code bad smells, and recommended refactorings. **(b)** Average ratings of the refactoring recommendations which are plotted in coarse-grained scales of negative, neutral, and positive along each axis; recommendations with a tied rating are placed arbitrarily within an area.

D. Threats to Validity

Our study represents an initial attempt to assess the impact of traceability-enabled refactoring on the overall just-in-time RE practice. We discuss here some of the most important factors that must be considered when interpreting the results.

The *construct validity* [38] of our study can be affected by two major threats. On the operational level, we measured appropriateness of recommended refactorings only qualitatively by surveying WDS developers. We were unable to evaluate the quantitative effect by interrupting the developers within their daily work, or to determine whether our approach would have a direct impact on the actual implementation (quality, speed, etc.) of the just-in-time requirements. On the conceptual level, our guiding premise is that the more requirements implemented the better. While this may not be desired for processes like up-front RE, we believe it fits well with the lightweight nature of just-in-time RE which lacks "big picture" thinking [4]. One may think it is careless planning to even start implementing the features that are redundant or of little interest to the stakeholders, but we argue that oftentimes we do not know what features become irrelevant and removable until they are delivered.

One of the *internal validity* [38] threats relates to the sequential examination of the two steps in our approach, i.e., locating the "where" followed by selecting the "what". While this order is in line with the 6 refactoring activities defined by Mens and Tourwé [8] (cf. Section II-B), caution must be taken in interpreting our findings as a whole rather than separately. In particular, because the automated tracing results did not reach the 100% recall and precision (cf. Table III), it is not known how a perfectly traced set of code artifacts would affect the refactoring recommendations and their ratings. It is also not known how incorrect and imprecise traceability may have affected our current results. Another confounding variable stems from our design of basing the subjective rating on the targeted requirements. While all the three WDS developers in our study provided justifications directly related to REQ_1-REQ_5, carrying out an experiment with actual requirements implementation tasks can mitigate this threat to a great extent.

An important threat to *external validity* [38] is that our study was conducted on a single dataset. For requirements-to-source-code tracing, our approach relies upon the clustering-based link retrieval which is shown to perform accurately in several, but not all, project environments [13]. For refactoring recommendation, it is encouraging to note that our semantic categorization is in general applicable to the open source Mozilla and the proprietary WDS projects. We therefore position our study as a proof-of-concept stepping stone toward more rigorous analyses and assessments.

V. Related Work

Software traceability research has its root in Gotel and Finkelstein's seminal work [22]. A majority of the effort has been spent in *recovering* the traceability links [30]. Our work advances the literature by showing that traceability can be more than recovered. It can be *utilized*. Research in this direction includes using traceability to support software change tasks [39] and examining trace granularity (requirement-to-class vs. requirement-to-method) from a cost-benefit perspective [25]. Our work focuses on a different application area: code refactoring. Interestingly, we support the finding that class-level traces are more practically valuable [25] because our approach can recommend refactorings at both class and method levels (cf. Table I).

Utilizing traceability does not have to wait till the end of the project. In [40, 41], Cleland-Huang *et al.* proposed event-based traceability to generate timely project notifications by monitoring changes made to software artifacts. We also built a monitor to promptly remind the developer of resolving the code smells at their early appearance [11]. Our current framework thus institutes a "monitor" in Fig. 3 to allow for instant refactoring recommendations. We plan to combine the requirements-driven and monitor-based refactoring mechanisms in the next release of InsRefactor.

Tahvildari and Kontogiannis [42] were among the first to use requirements to guide software re-engineering. They encoded softgoal interdependency graphs to reason about code transformation's (partial) satisfaction of the quality goals like performance and maintainability. Our work complements theirs by leveraging both functional and quality requirements to guide code transformation. A recent endeavor explored how different types of refactoring might help or hurt traceability [43]. In a word, our research differs from and also helps close the loop of that endeavor: The work in [43] is about "refactoring for traceability" and ours is about "traceability for refactoring".

VI. Conclusions

In this paper, we have contributed a novel traceability-enabled refactoring approach to addressing the challenges of just-in-time RE. Our approach takes full advantage of the requirements traceability information to identify the to-be-refactored locations and to rationalize why a certain type of refactoring should be carried out. A proof-of-concept study on an industrial software system shows our approach's accuracy of locating "where" to refactor and precision of recommending "what" to refactor.

Our overarching goal is to improve traceability and refactoring practices by fully exploiting their interconnections. Our future work includes enriching the requirements-driven scheme for refactoring, pushing the relevant recommendations instantly to the stakeholders, and maintaining traceability after refactorings are performed.

Acknowledgement

We are grateful to the partner company for the generous support of our research and Zhangji Chen for assisting in data analysis. The work is in part supported by the U.S. NSF (National Science Foundation) Grant CCF-1238336. Authors from Beijing Institute of Technology are supported by the National Natural Science Foundation of China (No. 61272169), Program for New Century Excellent Talents in University (No. NCET-13-0041), and Beijing Higher Education Young Elite Teacher Project (No. YETP1183).

References

[1] J. Aranda, S. Easterbrook, and G. Wilson, "Requirements in the wild: how small companies do it," in *RE*, 2007, pp. 39–48.

[2] T. A. Alspaugh and W. Scacchi, "Ongoing software development without classical requirements," in *RE*, 2013, pp. 165–174.

[3] M. Lee, "Just-in-time requirements analysis — the engine that drives the planning game," Agile Alliance, Technical Report, May 2002.

[4] N. A. Ernst and G. C. Murphy, "Case studies in just-in-time requirements analysis," in *EmpiRE*, 2012, pp. 25–32.

[5] E. Knauss, D. Damian, G. Poo-Caamaño, and J. Cleland-Huang, "Detecting and classifying patterns of requirements clarifications," in *RE*, 2012, pp. 251–260.

[6] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

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

[8] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE TSE*, vol. 30, no. 2, pp. 126–139, February 2004.

[9] E. Murphy-Hill and A. P. Black, "Refactoring tools: fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, Sept/Oct 2008.

[10] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE TSE*, vol. 38, no. 1, pp. 5–18, January/February 2012.

[11] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE TSE*, vol. 39, no. 8, pp. 1112–1126, August 2013.

[12] H. Liu, Y. Gao, and Z. Niu, "An initial study on refactoring tactics," in *COMPSAC*, 2012, pp. 213–218.

[13] N. Niu and A. Mahmoud, "Enhancing candidate link generation for requirements tracing: the cluster hypothesis revisited," in *RE*, 2012, pp. 81–90.

[14] S. Nair, J. L. de la Vara, and S. Sen, "A review of traceability research at the requirements engineering conference," in *RE*, 2013, pp. 222–229.

[15] B. Boehm, "Software engineering," *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 1226–1241, December 1976.

[16] W. Scacchi, "Understanding the requirements for developing open source software systems," *IEE Software*, vol. 149, no. 1, pp. 24–39, February 2002.

[17] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[18] P. Heck and A. Zaidman, "An analysis of requirements evolution in open source projects: recommendations for issue trackers," in *IWPSE*, 2013, pp. 43–52.

[19] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *FSE*, 2012, article no. 50.

[20] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, April 2011.

[21] M. Cusumano and R. Selby, *Microsoft Secrets*. Free Press, 1998.

[22] O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in *ICRE*, 1994, pp. 94–101.

[23] K. Pohl, "PRO-ART: enabling requirements pre-traceability," in *ICRE*, 1996, pp. 76–85.

[24] J. H. Hayes, A. Dekhtyar, and J. Osborne, "Improving requirements tracing via information retrieval," in *RE*, 2003, pp. 138–147.

[25] A. Egyed, F. Graf, and P. Grünbacher, "Effort and quality of recovering requirements-to-code traces: two exploratory experiments," in *RE*, 2010, pp. 221–230.

[26] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. I. Maletic, and P. Mäder, "The quest for ubiquity: a roadmap for software and systems traceability research," in *RE*, 2012, pp. 71–80.

[27] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *IEEE TSE*, vol. 32, no. 1, pp. 4–19, January 2006.

[28] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *ICPC*, 2010, pp. 68–71.

[29] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *ICPC*, 2012, pp. 183–192.

[30] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability," *Empirical Software Engineering*, (accepted).

[31] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "Automated classification of non-functional requirements," *Requirements Engineering*, vol. 12, no. 2, pp. 103–120, April 2007.

[32] S. Liaskos, A. Lapouchnian, Y. Yu, E. S. K. Yu, and J. Mylopoulos, "On goal-based variability acquisition and analysis," in *RE*, 2006, pp. 76–85.

[33] N. Niu and S. Easterbrook, "Extracting and modeling product line functional requirements," in *RE*, 2008, pp. 155–164.

[34] C. Fillmore, "The case for case," in *Universals in Linguistic Theory*, E. Bach and R. Harms, Eds. New York: Holt, Rinehart and Winston, 1968, pp. 1–88.

[35] M. P. Robillard, R. J. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, July/August 2010.

[36] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, (accepted).

[37] J. L. Fleiss, B. Levin, and M. C. Paik, *Statistical Methods for Rates and Proportions*. Wiley-Interscience, 2003.

[38] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[39] P. Mäder and A. Egyed, "Assessing the effect of requirements traceability for software maintenance," in *ICSM*, 2012, pp. 171–180.

[40] J. Cleland-Huang, C. K. Chang, and M. J. Christensen, "Event-based traceability for managing evolutionary changes," *IEEE TSE*, vol. 29, no. 9, pp. 796–810, September 2003.

[41] J. Cleland-Huang, P. Mäder, M. Mirakhorli, and S. Amornborvornwong, "Breaking the big-bang practice of traceability: pushing timely trace recommendations to project stakeholders," in *RE*, 2012, pp. 231–240.

[42] L. Tahvildari and K. Kontogiannis, "A software transformation framework for quality-driven object-oriented re-engineering," in *ICSM*, 2002, pp. 596–605.

[43] A. Mahmoud and N. Niu, "Supporting requirements traceability through refactoring," in *RE*, 2013, pp. 32–41.