On the Effectiveness of Accuracy of Automated Feature Location Technique

Takashi Ishio*, Shinpei Hayashi†, Hiroshi Kazato‡, Tsuyoshi Oshima§
*Osaka University, Osaka 565–0871, Japan Email: ishio@ist.osaka-u.ac.jp
†Tokyo Institute of Technology, Tokyo 152–8552, Japan Email: hayashi@se.cs.titech.ac.jp
‡NTT DATA INTELLILINK CORPORATION, Tokyo 104–0052, Japan Email: kazatoh@intellilink.co.jp
§NTT Software Innovation Center, Tokyo 180–8585, Japan Email: oshima.tsuyoshi@lab.ntt.co.jp

***Abstract*—Automated feature location techniques have been proposed to extract program elements that are likely to be relevant to a given feature. A more accurate result is expected to enable developers to perform more accurate feature location. However, several experiments assessing traceability recovery have shown that analysts cannot utilize an accurate traceability matrix for their tasks. Because feature location deals with a certain type of traceability links, it is an important question whether the same phenomena are visible in feature location or not. To answer that question, we have conducted a controlled experiment. We have asked 20 subjects to locate features using lists of methods of which the accuracy is controlled artificially. The result differs from the traceability recovery experiments. Subjects given an accurate list would be able to locate a feature more accurately. However, subjects could not locate the complete implementation of features in 83% of tasks. Results show that the accuracy of automated feature location techniques is effective, but it might be insufficient for perfect feature location.**

***Index Terms*—feature location, impact analysis, program comprehension, human factor**

I. Introduction

Feature location is a program understanding phase in software maintenance. A *feature* represents a functionality that is defined by requirements and which is accessible to developers and users [1]. Locating the current implementation of a feature in source code is important for developers to perform various maintenance tasks such as enhancement, bug fixing, and refactoring related to the feature. Mäder *et al.* reported that developers who know source files related to requirements can produce a software change more efficiently [2]. Therefore, feature location in this paper denotes a phase to find code snippets relevant to a feature to the greatest extent possible before source code modification.

Although feature location is important, locating the complete implementation of a feature is difficult for developers [3], [4]. According to Wang *et al.* [3], given a feature, developers use a keyword search tool to identify "seed" methods that are likely to be relevant to the feature. For each seed method, developers explore its source code and related methods to validate whether the methods are actually relevant to the feature or not. Although each step in the process seems simple, developers must identify relevant methods before they can ascertain the complete implementation of a feature. Furthermore, industrial developers are often asked to locate features in an unfamiliar system using only its source code because an enterprise application might outlive its development team.

To support developers locating a feature, various automated feature location techniques have been proposed [1]. A typical technique takes keywords or a description of a feature as input and extracts a list of methods that are only relevant to the feature. For example, Marcus *et al.* [5] proposed an application of latent semantic indexing (LSI) to feature location. Poshyvanyk *et al.* [6] proposed the use of execution traces of a target program to improve a ranking obtained by LSI. Eaddy *et al.* [7] combined static analysis, dynamic analysis and an information retrieval (IR) technique to extract a better ranking. Developers can investigate source code using a result of these techniques, *e.g.* the top 10 methods of a ranking.

A more accurate result of automated feature location techniques is expected to enable developers to perform more accurate feature location. However, a question remains as to whether the accuracy of automated feature location techniques actually contributes or not to the performance of developers. This question arises from an observation reported by Cuddeback *et al.* [8], Kong *et al.* [9] and Dekhtyar *et al.* [10]. They asked analysts to do manual validation of a traceability matrix between requirements and system tests. The results indicated that, whereas analysts given a less-accurate traceability matrix can identify many false positives and false negatives in the matrix, analysts given an accurate traceability matrix tend to decrease the overall accuracy of the matrix. If the same phenomena are visible in feature location, then accurate automated feature location results might be less effective for actual feature location performed by developers.

To answer the question, we conducted an experiment with 20 subjects in three organizations. We asked the subjects to locate a feature using a given list of methods by excluding false positives from the list and by identifying false negatives in the source code. To evaluate the influence of accuracy of lists, we prepared a pair of tasks that specify the same feature to be located but which provide different lists of methods. The lists are derived artificially from the result of a textual search using LSI, so that the lists have different accuracy, but have the same length. The features and the result of LSI used in this experiment are involved in the open dataset of the work by Dit and Gethers [1], [11], although we have extended the descriptions of features. Each subject used either an accurate list or a less-accurate list for a feature.

Consequently, we obtained the following observations.

- Subjects given an accurate list located a feature more accurately than subjects given a less-accurate list.
- Subjects missed one or more relevant methods in 83 of 100 tasks. Even if an accurate list covered the complete implementation of a feature, several relevant methods have been falsely recognized as false positives.
- Subjects improved precision of the final result by excluding false positives. However, the subjects did not improve recall in several features.

Differently from the experiments on traceability recovery, the result shows that developers can utilize results of automated feature location techniques. However, developers tend to miss methods that are relevant to a feature if its description is not clear sufficient for developers.

The contributions of this paper are summarized as follows.

- We have shown the manner in which accuracy of automated feature location techniques contribute to the accuracy of feature location tasks performed by developers.
- We have reported the manner in which we have prepared feature location tasks from an existing benchmark. It is useful for researchers to conduct a similar experiment using the benchmark.
- We have made our dataset freely available so that other researchers can replicate the experiment[1].

The sections are organized as follows. Section II describes related work and our research background. We state our research questions in Section III. Section IV explains the setting of our experiment. Section V shows the results of the experiment. In Section VI, we discuss the observations and the threats to the validity of our experiment. Section VII describes conclusions and future work.

II. Related Work

In software maintenance, developers must understand the unfamiliar source code of a target system. Ko *et al.* [12] reported that developers often use a keyword search tool to identify source code relevant to their tasks. Wang *et al.* [3] emphasized the process of manual feature location in software maintenance. They also observed that developers often used a keyword search tool to identify "seed" methods that are likely to be relevant to features.

Several researchers reported that developers were unable to identify the complete implementation of features. Wang *et al.* [3] observed that each of recall and precision is less than 75% in their feature location tasks. Egyed *et al.* [4] conducted an experiment of manual recovery of requirements-to-code links. Compared with the correct links created by developers of the programs, subjects recognized 95% of irrelevant pairs of requirements and classes, whereas the subjects missed about half of the relevant pairs. Lindvall *et al.* [13] conducted a case study of manual impact analysis. The result shows that developers were able to predict only a half of classes to be modified for the next maintenance release.

Various automated feature location techniques have been proposed in the literature [1]. Some are comparable in recall and precision to manual feature location. SNIAFL [14] shows that its precision and recall are, respectively, 91% and 99% for several small programs. CERBERUS [7] shows that its precision and recall are, respectively, 75% and 73% in the best configuration for a particular set of concerns. Gethers *et al.* [11] compared several impact analysis techniques in more practical settings. The best analysis automatically identified 41–75% of methods modified for a feature request. Revelle *et al.* [15] manually evaluated results of several automated feature location techniques. They reported that the best technique identified 3 relevant methods among the top 10 methods.

Many researchers investigated whether developers can utilize a result of automated techniques or not. Revelle *et al.* [15] reported that the authors' and several students' validation results agreed over 90% of the time for a certain feature. The observation is promising but not generalizable because it is a single case and the authors have identified only true positives in the results rather than the complete implementation of features. In traceability research, Cuddeback *et al.* [8] investigated manual validation of the requirement traceability matrix representing links between requirements and system tests. They reported that analysts given an accurate traceability matrix decreased the overall accuracy of the matrix. Kong *et al.* [9] analyzed the process of the traceability validation tasks. Dekhtyar *et al.* [10] confirmed the observation by statistical analysis. The analysis indicates that developers cannot utilize accurate traceability links. Based on the results described above, Cuddeback *et al.* [16] discussed a means of addressing the inaccuracy of developers. Ghabi *et al.* [17] proposed automated validation for traceability recovery.

In automated debugging research, Parnin *et al.* [18] conducted a controlled experiment to evaluate the usefulness of automated debugging techniques. The experiment used an artificially modified ranking derived from a result of an automated technique. It showed that a ranking change did not affect the performance of developers. Chatterji *et al.* [19] reported that developers were not able to use code clone detection for bug fixing with no training in code clones. These research efforts also indicated the importance of human study to evaluate the actual usefulness of automated support for developers.

In our experiment, we used LSI to extract a list of methods relevant to a feature. LSI-based feature location was proposed by Marcus *et al.* [5]. Poshyvanyk *et al.* [6] combined dynamic analysis with LSI to improve ranking. Binkley *et al.* [20] showed that identifier normalization is effective to improve search results. The effectiveness of LSI was also confirmed in fault localization. Beard *et al.* [21] reported that LSI can recommend an appropriate starting point to locate a fault in source code for 60 out of 63 bugs in a system. To keep our experiment simple and easy to replicate, we have used a simple

[1]http://sel.ist.osaka-u.ac.jp/~ishio/FL/

TABLE I
Features, Tasks, and Their Accuracy

System	Feature (Issue ID)	# methods (Type)	Task (Type)	Precision	Recall	F-measure
jEdit	f_{J0} (2122926)	1	J_0 (example)	0.33 (1/3)	1.00 (1/1)	0.50
	f_{J1} (1747300)	13 (larger)	J_{1b} (better)	1.00 (10/10)	0.77 (10/13)	0.87
			J_{1w} (worse)	0.40 (4/10)	0.31 (4/13)	0.35
	f_{J2} (2668434)	6 (smaller)	J_{2b} (better)	0.60 (6/10)	1.00 (6/6)	0.75
			J_{2w} (worse)	0.30 (3/10)	0.50 (3/6)	0.38
	f_{J3} (1593464)	10	J_3 (goldset)	1.00 (10/10)	1.00 (10/10)	1.00
muCommander	f_{M0} (311)	2	M_0 (example)	0.33 (1/3)	0.50 (1/2)	0.40
	f_{M1} (60)	32 (larger)	M_{1b} (better)	1.00 (10/10)	0.31 (10/32)	0.48
			M_{1w} (worse)	0.80 (8/10)	0.25 (8/32)	0.38
	f_{M2} (231)	6 (smaller)	M_{2b} (better)	0.60 (6/10)	1.00 (6/6)	0.75
			M_{2w} (worse)	0.30 (3/10)	0.50 (3/6)	0.38

LSI approach that computes similarity among methods and the description of a feature [11].

III. Research Questions

We have conducted a controlled experiment on feature location using human subjects. We have defined a process of feature location supported by an automated technique as follows: given an *initial list* of methods produced by an automated technique, a developer *validates* the list by removing irrelevant methods (false positives) from the list and by adding missing relevant methods (false negatives) to the list. Ideally, a *validated* list should include no irrelevant methods but all methods relevant to a feature.

We formulated the following four research questions.

RQ1 *Do better initial recall and precision engender better performance in feature location by developers?*

RQ2 *Which option is more important for feature location, initial recall or precision?*

RQ3 *How do developers spend time to validate a list of methods?*

RQ4 *How does a validated list differ from its initial list?*

We carefully set up feature location *tasks* of several types; each task is validation of an initial list of methods in a limited time. To answer RQ1, we introduced two categories of tasks: *better* and *worse*. The initial lists of better tasks are accurate, *i.e.*, having better precision and recall values, than worse tasks. We compared the F-measure of the validated lists between those of better and worse tasks.

For a response to RQ2, we also introduced two categories of tasks: *precision-intensive* and *recall-intensive*. The initial lists of precision-intensive tasks have high precision but low recall; they are related to little false positives and many false negatives. In contrast, the initial lists of recall-intensive tasks have high recall but low precision. We compared F-measures of the validated lists between precision-intensive tasks and recall-intensive tasks.

To answer RQ3, we have analyzed the activities of subjects. Each subject is asked to input their judges in our Eclipse plug-in. We have analyzed a series of judges with timestamps to elucidate how they functioned and whether subjects had sufficient time or not.

To answer RQ4, we have analyzed the manner in which recall and precision are improved or degraded by validation tasks.

IV. Experimental Setup

A. Features

The features for our study were prepared using an existing change-history-based feature location benchmark [1]. This benchmark is constructed using the information of changes in revision control systems such as Subversion, and the related tickets in issue tracking systems such as Trac or Bugzilla. The benchmark provides a list of change requests including feature requests and bug reports. Each change request is associated with a list of methods modified to implement the change, or a *goldset*. For objectivity, we have used the existing dataset.

The seven extracted features are shown in the left columns of Table I. The symbol of a feature is used to identify the feature in the remaining part of the paper. The Issue ID of a feature indicates the identifier of the feature in the benchmark.

We have selected five features for the experiment from two systems: muCommander and jEdit. We extracted a *larger* feature and a *smaller* feature from each system to extract precision-intensive and recall-intensive tasks. f_{J1} and f_{M1} are larger features, of which goldsets have more than 10 methods. f_{J2} and f_{M2} are smaller features, of which goldsets have fewer than 10 methods. The feature f_{J3} is extracted for checking the cases in which subjects receive the true goldset having just 10 methods.

In addition to these five features, we have selected two tiny features: f_{J0} implemented in a single method and f_{M0} implemented in two methods. These features are used to explain the process of feature location to subjects.

To use the benchmark for our experiment, we fixed two problems in the benchmark as follows:

- **Curation of goldsets**. Some goldsets of the benchmark include inappropriate methods. Because a goldset of a feature is generated automatically from a change set of source code in a revision history, it includes methods that are irrelevant to the target feature if developers commit multiple intentional changes at once, *e.g.*, the implementation of the feature together with a refactoring. Additionally, if rename refactorings are performed after

muCommander Feature #231

Short Description
"Skip all" for errors that occur during a file transfer operation

Long Description
As suggested in the [http://www.mucommander.com/forums/viewtopic.php?f=2&t=938 forums]: adding a "Skip all" button when an error occurs in a multiple file move / copy operation would be a nice feature to have.

Feature Description
muCommander has a feature that copies/moves files selected by a user. When a user tries to execute a copy, muCommander shows a dialog to specify a destination directory. Pushing the Copy button in the dialog starts a copy process. If an error occurred during the copying of a file, then an error dialog shows a message and asks the user to skip the file, retry to copy the file, or cancel the copy process.
The new feature is "Skip All." The dialog to specify a destination directory has a new check box with the caption is "Skip errors." If a user checked the box, then muCommander automatically skips a file if an error occurred, without showing a dialog. The error message dialog also has a new button "Skip all." If the button is pushed, then muCommander shows no error dialog in further errors, as "Skip errors" is checked.

Fig. 1. Descriptions of feature f_{M2}.

implementing the feature, then some methods in the goldset might no longer be found in the source code. We have manually identified renamed methods and updated such goldsets. It is noteworthy that the number of methods in Table I reflects this manual refinement.

- **Addition of extended descriptions**. Some descriptions in the benchmark do not describe the associated features accurately. Because descriptions are submitted to an issue tracking system by a requester of the feature, a gap separates the description and the actually implemented feature. So that subjects can correctly understand features from their descriptions, we provided an extended description for each feature in addition to the original description. An example of the extended description is presented as "Feature Description" in Figure 1. For all extended descriptions, we used a single paragraph to describe the basic behavior without the associated feature and another paragraph to describe the target feature to be located. The extended descriptions are based on the recorded changes in the benchmark. We tried to explain all the recorded changes although the original descriptions do not mention some of them.

Each feature comprises the following elements: (1) a short description in the benchmark, (2) a long description in the benchmark, (3) an extended description, (4) a screenshot of an execution, and (5) the goldset.

B. Tasks

From the extracted features *tasks* are generated. A task is a pair of a feature and an initial list of methods for the feature. The goal of a task is to validate the initial list using source code of a system and the descriptions of the feature. To normalize the effort of subjects, every task is associated with 10 methods, except for example tasks.

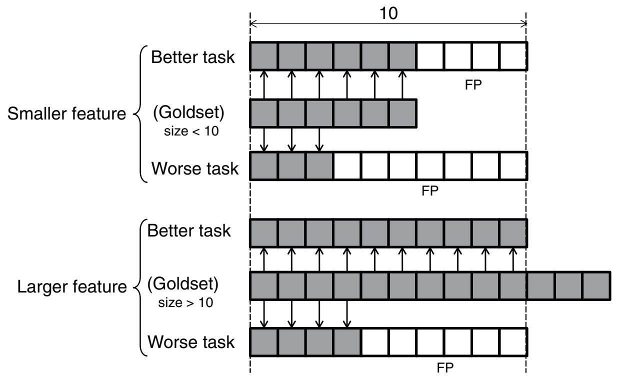


Fig. 2. Generation of *better* and *worse* tasks.

We evaluate the performance of a task using precision, recall, and F-measure for the initial list and the validated list. Letting g, i and t be the size of a goldset, an initial list, and true positives, respectively, then the initial F-measure is given by the harmonic mean of precision t/i and recall t/g:

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall} = \frac{2t}{g+i}.$$

By eliminating t using $2t = F \cdot (g+i)$, precision and recall can be written in terms of F, i and g.

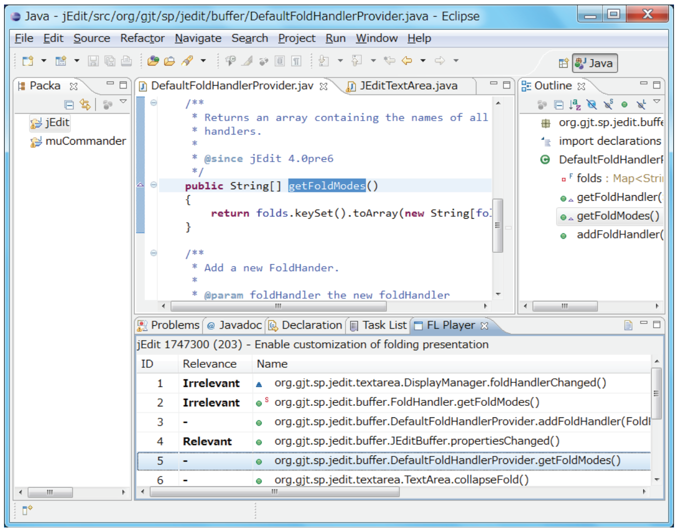
$$\begin{aligned} precision &= \frac{F \cdot (g+i)}{2i} = \frac{F}{2} \cdot \left\{1 + \frac{g}{i}\right\} \\ recall &= \frac{F \cdot (g+i)}{2g} = \frac{F}{2} \cdot \left\{1 + \frac{i}{g}\right\} \end{aligned}$$

In the experiment, we controlled F as an independent variable and fixed i at 10. Then the tradeoff between precision and recall is determined by g, where precision gets better for larger features and recall does better for smaller ones.

We have generated *better* and *worse* tasks for each feature. The lists in better tasks include methods relevant to the features (*i.e.*, *gold methods*) to the greatest extent possible, whereas the lists in worse tasks include several methods that are irrelevant to the features (*false positives*). A notational example is shown in Figure 2 to illustrate how a pair of tasks is generated from the goldset of a feature. In the figure, a gray box and a white box respectively represent a gold method and a false positive. For smaller features having fewer than 10 gold methods, generated better tasks have all the gold methods and some false positives. In contrast, for larger features having more than 10 gold methods, generated better tasks have 10 gold methods. The unused gold methods are regarded as false negatives. Generated worse tasks have some gold methods and false positives. We have injected false positives into worse tasks until their F-measures become less than the threshold F_T. We use $F_T = 0.40$ to differentiate the number of false positives in better and worse tasks clearly.

TABLE II
Task Assignment

Subject	Organization	1st	2nd	3rd	4th	5th
1	Osaka University	J_{1b}	J_{2b}	M_{1w}	M_{2w}	J_3
2		J_{2b}	J_{1w}	M_{1b}	M_{2w}	J_3
3		J_{2w}	J_{1b}	M_{2b}	M_{1w}	J_3
4		J_{1w}	J_{2w}	M_{1b}	M_{2b}	J_3
5		M_{2b}	M_{1b}	J_{1w}	J_{2w}	J_3
6		M_{1b}	M_{2w}	J_{2b}	J_{1w}	J_3
7		M_{1w}	J_{2b}	J_{2w}	J_{1b}	J_3
8		M_{2w}	M_{1w}	J_{2b}	J_{1b}	J_3
9	Tokyo Institute of Technology	J_{2b}	J_{1b}	M_{2w}	M_{1w}	J_3
10		J_{1b}	J_{2w}	M_{1w}	M_{2b}	J_3
11		J_{1w}	J_{2b}	M_{1b}	M_{2w}	J_3
12		J_{2w}	J_{1w}	M_{1b}	M_{2b}	J_3
13		M_{1b}	M_{2b}	J_{2w}	J_{1w}	J_3
14		M_{2b}	M_{1w}	J_{2w}	J_{1b}	J_3
15		M_{2w}	M_{1b}	J_{1w}	J_{2b}	J_3
16		M_{1w}	M_{2w}	J_{2b}	J_{1b}	J_3
17	NTT	J_{1b}	J_{2w}	M_{2b}	M_{1w}	J_3
18		J_{2b}	J_{1w}	M_{2w}	M_{1b}	J_3
19		M_{1b}	M_{2b}	J_{1w}	J_{2w}	J_3
20		M_{2w}	M_{1w}	J_{1b}	J_{2b}	J_3



```
    /**
     * Returns an array containing the names of all
     * handlers.
     *
     * @since jEdit 4.0pre6
     */
    public String[] getFoldModes()
    {
        return folds.keySet().toArray(new String[fo
    }

    /**
     * Add a new FoldHander.
     *
     * @param foldHandler the new foldHandler
```

jEdit 1747300 (203) - Enable customization of folding presentation

ID	Relevance	Name
1	**Irrelevant**	org.gjt.sp.jedit.textarea.DisplayManager.foldHandlerChanged()
2	**Irrelevant**	org.gjt.sp.jedit.buffer.FoldHandler.getFoldModes()
3	-	org.gjt.sp.jedit.buffer.DefaultFoldHandlerProvider.addFoldHandler(Foldl
4	**Relevant**	org.gjt.sp.jedit.buffer.JEditBuffer.propertiesChanged()
5	-	org.gjt.sp.jedit.buffer.DefaultFoldHandlerProvider.getFoldModes()
6	-	org.gjt.sp.jedit.textarea.TextArea.collapseFold()

Fig. 3. Screenshot of FLPlayer.

The precision, recall and *F*-measure of the initial lists of the generated tasks are shown in the right columns of Table I. The tasks generated from larger features include a few false positives but fail to capture many of gold methods. In contrast, the tasks generated from smaller features cover almost all of their goldsets, but include many false positives. Therefore, the former and latter tasks are regarded respectively as *precision-* and *recall-intensive*.

We have used LSI to select gold methods and false positives included in the initial lists. Instead of computing LSI result by ourselves, we used the LSI result prepared by Gethers *et al.* [11], which is an extension of the benchmark for impact analysis. In Gethers's dataset, an LSI result for a feature is a ranking of methods with contents similar to the long description of the feature. We have selected an appropriate number of gold methods and false positives from the top of the ranking. For example, J_{1w} includes a list of four gold methods and six false positives. The gold methods have higher similarity than the nine other relevant methods. The false positives have higher similarity than other irrelevant methods. The generated initial lists are sorted by their LSI ranking.

C. Subjects and Task Assignment

We recruited 20 subjects from three organizations of both academia and industry. Subjects included 16 students of software engineering and 4 industrial developers. Their Java experience was widely distributed from zero to 16 years, with a median of three years.

No subject knew the target systems. This situation is common in software maintenance tasks. Developers might have to update legacy software developed by other teams.

Each subject examined the five assigned tasks in the order presented in Table II. We carefully assigned the tasks for each subject, satisfying the following constraints:

- Every subject examines the first and second tasks of the same system followed by the third and fourth tasks of the other system, and ends with the goldset task (J_3).
- Every subject covers all of the five features.
- Every subject experiences all of the different types of tasks: a better precision-intensive task, a worse precision-intensive task, a better recall-intensive task, and a worse recall-intensive task.
- Every task is examined by at least 10 subjects.

D. Environment

Subjects are given an Eclipse IDE installed with a special view named FLPlayer. We chose Eclipse as our environment because of its publicity and familiarity to subjects. FLPlayer is a view for validating a list of methods in Eclipse. A screenshot of the view is presented in Figure 3. Subjects can see a list of methods as a table in the view. When they double-click a method in the view, a source code editor automatically opens and moves to the definition of the method. After investigating the source code, they answer whether the method is *relevant* or *irrelevant* to the feature, using the drop-down menu in the second column of the view. For example, in the figure, at least six methods are enumerated. Two of them are specified as irrelevant. In addition, subjects can add a method that is not listed in the view from the context menu of the method in a source code editor.

For each task, every subject is given a printed document for a task and Eclipse environment in which FLPlayer is showing the list of methods ordered by their LSI ranking. The document included the descriptions of a feature, a screenshot, the same list of methods in the same order as shown in FLPlayer, and a quick reference guide of Eclipse and FLPlayer. The subjects are not allowed to access other online materials such as an issue tracking system. We did not provide the LSI scores to subjects and used them just for the order of methods because they might reveal which methods are injected as false positives.

Eight laptop computers of the same model were used for the experiment, each equipped with Core i5 processor (Intel Corp.), 4 GB RAM, 256 GB SSD and a 12.1-inch LCD monitor with WXGA (1280×800) resolution. They execute Eclipse 3.7.2 on Windows 7 (Microsoft Corp.) and JDK 1.7.0 without a network connection. Although each laptop has an embedded pointing device, subjects were allowed to bring their favorite devices.

E. Procedure

We operated the experiment three times: once for each organization. For the convenience of the subjects, each operation is conducted at the location where they belong, using the same instruments and procedure.

At the beginning of the operation, subjects were given the following introduction in an hour, using a PowerPoint presentation: (1) the purpose of the experiment and the goal of feature location, (2) usage of Eclipse, *e.g.*, showing hierarchies of method calls and class inheritance relations, searching references from/to a method, etc., (3) introduction to the FLPlayer plug-in, (4) an exercise in muCommander using the task M_0, and (5) another exercise in jEdit using the task J_0.

Ten minutes were given for each exercise. Then the answer and the reason were explained. A method is relevant to a feature if at least a single line of code in the method is necessary to execute the feature, according to the definition of goldsets in the benchmark. Although we initially arranged the same configurations of the laptops for all subjects, they were allowed to change them during the exercises to fit their preferences. They were told that we close all editors in Eclipse between sessions, but we do not change their preferences settings.

Five sessions were conducted after the introduction. In each session, the subjects performed a task in 30 min. At the end of the session, they filled in a questionnaire to answer whether they were able to understand the task, were given sufficient time, and were confident with their answers. Between sessions, they were asked to leave the room and have a break of about 10 minutes. We set up instruments for the next session during that interval. After finishing all sessions, we asked the subjects to fill in the questionnaire.

V. Results

This section presents a discussion of the results of our experiment to answer the research questions formulated in Section III. In this section, *initial F-measure* and *validated F-measure* respectively denote F-measures of the initial lists of tasks given to the subjects and of the resultant lists validated by the subjects.

A. RQ1

In the experiment, each subject performed four tasks, two of which start with better initial accuracy than the others. We obtained 40 samples for each of *better* and *worse* tasks. Figure 4 is a box plot of validated F-measure of better and worse tasks. The task pairs are located on the horizontal axis, where the white and gray boxes respectively correspond to better and worse initial F-measure values. For all features, we observed that manual validation of better tasks tends to outperform worse ones.

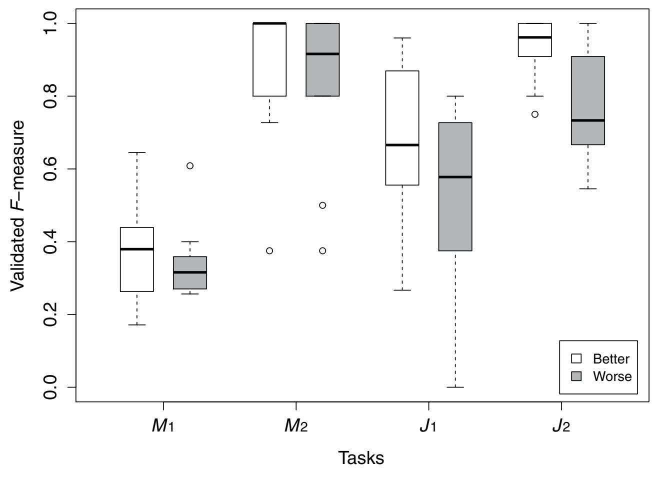


Fig. 4. Comparison of validated F-measure between *better* and *worse* tasks.

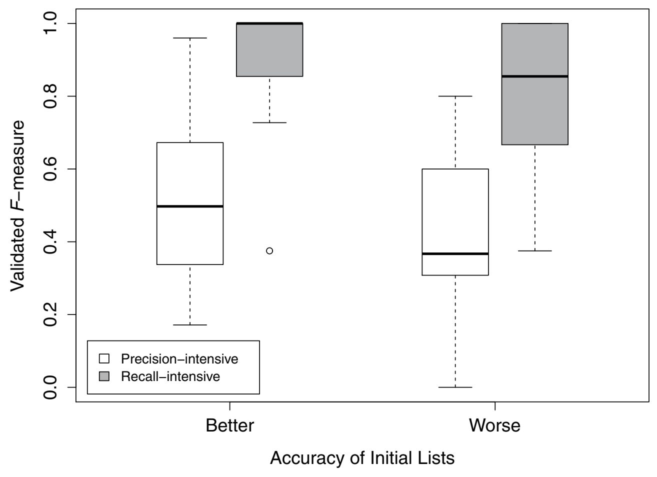


Fig. 5. Comparison of validated F-measure between precision-intensive and recall-intensive tasks.

To determine the performance of two kinds of tasks that are significantly different, we performed paired t-tests on the validated F-measure. The null hypothesis $H_0^{b \le w}$ and the alternative hypothesis $H_1^{b>w}$ are formulated as described below.

- $H_0^{b \le w}$: The average of validated F-measure of better tasks is equal to or less than that of worse tasks.
- $H_1^{b>w}$: The average of the validated F-measure of better tasks is greater than that of worse tasks.

The p-value was 0.008876, which is sufficiently small to reject $H_0^{b \le w}$ at the 1% significance level. We conclude that a statistically significant difference exist in final F-measure between better and worse tasks. Subjects given a more accurate list performed more accurate feature location.

It is noteworthy that the validated F-measure is improved

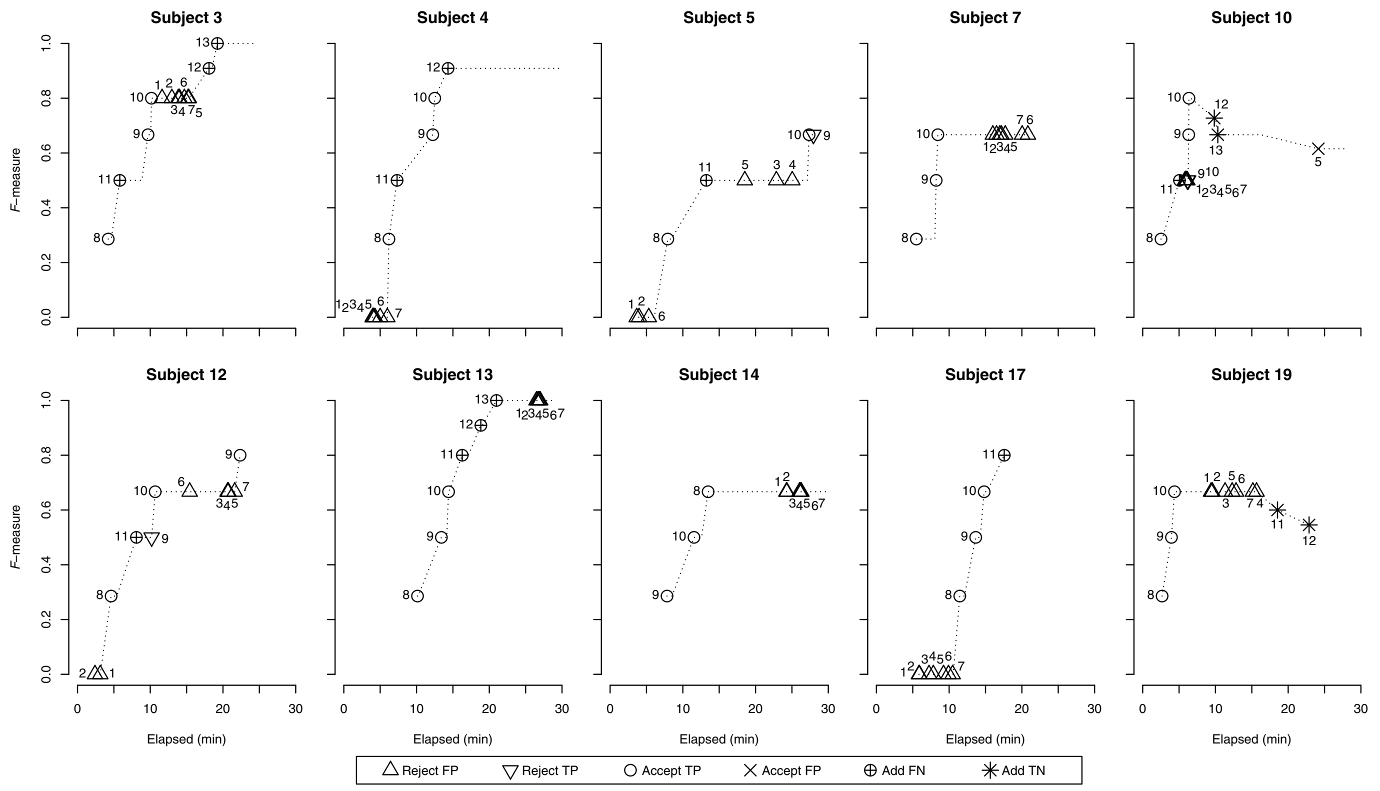


Fig. 6. Growth of accuracy during manual validation of J_{2w}.

from the initial F-measure for all tasks except for task M_{1b} and J_3. The effect of validation is discussed in Section V-D.

B. RQ2

We split samples of better and worse tasks into two categories: *recall-intensive* and *precision-intensive*. As a result, 20 samples were obtained for each combination of accuracy and size of goldset. Figure 5 shows a box plot of validated F-measure that compares the two categories. In both better and worse tasks, we observed that tasks with recall-intensive goldsets tend to outperform those with precision-intensive ones in the validated F-measure. It is noteworthy that our results show that 12 participants completely identified a feature in one or two tasks (17 tasks in total). All are recall-intensive tasks.

To determine whether the performance of tasks are significantly different or not, we formulated the null hypothesis $H_0^{p=r}$ and the alternative hypothesis $H_1^{p<r}$ as shown below.

- $H_0^{p=r}$: For the tasks with the same level of initial F-measure, the median of validated F-measure of precision-intensive tasks is the same as that of recall-intensive ones.
- $H_1^{p<r}$: For the tasks with the same level of initial F-measure, the median of validated F-measure of precision-intensive tasks is less than that of recall-intensive ones.

We performed the Wilcoxon signed-rank test to the median of validated F-measure for the two categories of tasks. In both tasks, the p-value was 9.537×10^{-07}, which is sufficiently small to reject $H_0^{p\geq r}$ at the 1% significance level. We conclude that a statistically significant difference exists in final F-measure between the categories. In other words, the initial recall is more important than the initial precision.

C. RQ3

All subjects were able to classify all methods in initial lists as either relevant or irrelevant in 30 min. We have analyzed how subjects spent time for their tasks using the timestamps of events recorded in FLPlayer. As a result, no significant difference was found among subjects because of their different backgrounds. No significant difference was found among tasks. Most subjects finished their tasks in 20 min.

Figure 6 includes time plots showing evolution of F-measure values for each subject performing task J_{2w}. The numbers shown indicate the rank of methods in the initial list. The added methods by subjects have numbers more than 10. The goldset of this task consists of six methods. The initial list of methods consists of three true positives and seven false positives. Subjects must find three false positives in the source code. Because the median value of the goldset size in the dataset of Gethers [11] is six, we chose J_{2w} as a representative task for locating feature enhancement requests. The following six kinds of symbols are put on vertices corresponding to operations of the subjects.

- △, ▽: rejection of false and true positives
- ◯, ×: acceptance of true and false positives
- ⊕, ✳: addition of false and true negatives

Although final F-measure values varied among subjects, all obtained final F-measure values better than those of initial

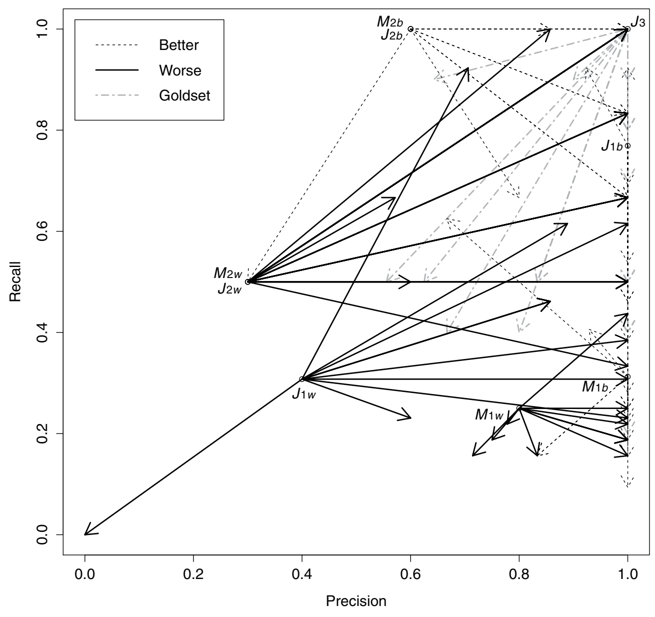


Fig. 7. Change in precision and recall.

ones ($F_{initial}$ = 0.38) within 20 min. All of them found at least two gold methods in the first 15 min. In addition, seven subjects (Subjects 3, 4, 5, 10, 12, 13 and 17) selected one or more false negatives. It is noteworthy that the ◯ and ⊕ symbols tend to appear contiguously, which indicates that when the subjects find one relevant method, they can correctly distinguish true positives around the method, and can sometimes even pick up false negatives.

Similarly, the △ symbols tend to appear contiguously in the time plots. The subjects identified all seven false positives, except Subject 5, who missed only one. It is noteworthy that few ▽ symbols appear in the series, indicating that the subjects can separate false positives from the list without losing true positives.

A similar tendency is observed in the time plots of other tasks, but we had to omit them from this paper because of length limits.

D. RQ4

Figure 7 shows the degrees of improvement in precision and recall for all treatments. Dashed, solid and dot-dash lines respectively correspond to better, worse and goldset tasks. It is readily apparent that most arrows are moving rightward, for improving precision, remarkably in worse tasks. This observation differs from the experiment conducted by Cuddeback *et al.* [8], who confirmed a tendency that arrows move to a diagonal line, *precision* = *recall*.

Precision was retained or improved in all tasks by most subjects. The subjects created the precise (precision 1.00) result in 68 of 100 tasks. In 82 tasks, precision is greater than 0.8. The result indicates that subjects can exclude many false positives from the initial lists.

However, recall has not been improved so much. For example, false negatives of M_{1b} and M_{1w} are not identified at all except for a few subjects. This is true partly because the feature is implemented by an abstract method and its many concrete implementations. The abstract method is likely to be sufficient to achieve the feature. The subjects did not identify all concrete implementations.

In several tasks, many subjects exhibited decreased recall. In M_{2b}, several subjects falsely recognized one or two relevant methods as irrelevant. Because each subject has chosen different methods as irrelevant, voting by several subjects can eliminate these errors. However, we have observed that a different understanding of a feature easily prevents developers from recognizing some true positives as follows.

In J_{1b} and J_3, many subjects falsely recognized a particular group of relevant methods as false positives. The task J_{1b} asked developers to locate a feature that enabled users to choose a shape of a marker shown in a window. Whereas the previous implementation has a triangle marker, the new feature supported new shapes such as a box and circle. The feature is implemented as a new item to select a marker shape in a preferences dialog and a new class `ShapedFoldPainter` for drawing new marker shapes. However, 7 of 10 subjects recognized the methods in the new class as irrelevant, perhaps because they understood that the feature simply modified the preferences dialog.

In J_3, the task described that the new feature enabled users to choose file icons for a view from operating system icons or default icons provided by jEdit. As a result, 13 of 20 subjects recognized several `getDefaultIcon` methods as irrelevant because they were likely to have been basic behavior. However, the methods are involved in the goldset because the concept of “default icon” has been introduced for the feature. In other words, the method names reflected the new feature, whereas their source code implements the basic behavior.

In M_{1b} and M_{1w}, the original description included an ambiguous phrase “the preferences dialog.” It refers to class `PreferenceDialog` and its subclasses `GeneralPreferencesDialog` and `ThemeEditorDialog`. Also, 6 of 20 subjects falsely recognized a method of class `ThemeEditorDialog` as a false positive because the dialog class name differs from the other two classes.

VI. Discussion

A. Accuracy of Automated Feature Location

We have shown that a more accurate list of methods enabled subjects to perform more accurate feature location (RQ1). That result emphasizes the importance of further research efforts to improve the accuracy of automated feature location techniques. Although a tradeoff exists between precision and recall, developers manually improved precision rather than recall (RQ2). As a consequence, a feature location technique with higher recall is a more promising direction. In traceability recovery experiments [8], [9], [10], no significant difference was found between recall and precision. One possible reason for this result is that subjects can understand features in detail during their tasks by reading relevant methods in initial lists. Because relevant methods are often connected by method call relations, more relevant methods in an initial list might provide a connected call graph that is easier to understand. However, system tests can be independently by natural language. Therefore, validating a link in a traceability matrix might not provide additional information about the manner in which requirements and system tests are related mutually.

Higher recall might be achieved by a longer list of methods, as evaluated in [11], [22]. In this experiment, developers validated 50 methods during five sessions. In each session, developers took about 20 min to validate 10 methods. If automated feature location techniques generate a longer list of methods, then both learning effects and fatigue might affect the performance of developers. To utilize a longer list, an additional support such as a keyword search for a list of methods would be needed, as Parnin *et al.* suggested for automated debugging [18].

Although subjects located a complete implementation of a feature in 17 of 100 tasks, many subjects falsely recognized relevant methods as false positives. One reason is that a target feature in a program is dependent on another feature in the program. The goldsets included such methods in the dependent feature because they are also modified to implement the target feature. However, subjects showed difficulty determining whether such methods should be a part of a feature or not. Allowing developers to categorize methods into three categories as relevant, irrelevant, and marginal might be effective to avoid the problem.

An insufficient description of a feature also prevents developers from accurate feature location. Each description of a feature request recorded in an issue tracking system often excludes the basic behavior of software without the feature because the description is written by users or developers who know other basic features of the software. To conduct our experiment, we must manually extend the description of a feature to enable subjects to distinguish the target feature from basic features.

A clearer understandable description of a feature is important but challenging because a description might become ambiguous if developers modified source code. For example, one presumes a description that explicitly refers to classes by their names, *e.g.* `PreferencesDialog` instead of a phrase "the preferences dialog." If a developer added a new subclass of the class, the developer must inspect the description and the feature implementation to ascertain whether the description should be updated or not. This problem is similar to *fragile pointcut problem* [23]. A pointcut is a predicate to identify program elements in Aspect-Oriented Programming [24]. A pointcut is fragile because a change in source code accidentally affects a set of program elements selected by the pointcut. A feature description might be more fragile because it is written in natural language, whereas pointcut is a formal predicate. As Mäder *et al.* [25] proposed for analysis of developers' activities affecting traceability links, some technique to maintain consistency between feature description and source code might be very useful.

Another means to improve recall is the use of interactive feature location techniques such as [26], [27]. They can recommend methods related to a method by which developers focus so that developers can identify more false negatives in source code. If the tools can provide an explanation of why methods are recommended as relevant, then the tools might be effective to avoid the accidental exclusion of true positives.

B. Threats to Validity

1) Internal Validity: The first threat to internal validity is related to the feature location technique we used for the experiment. To obtain lists of methods and to control their accuracy artificially, we exploit the result of IR-based feature location technique taken from the dataset of Gethers [11]. Because comparison of feature location techniques by precision and recall is a common means of evaluation in the literature, we expect that the IR-based technique can be replaced with another as long as its result has comparable precision and recall.

However, given a feature location technique for prioritizing methods that are difficult for developers to find, then the result might be inverted. For example, De Lucia [28] conducted an experiment on labeling classes and pointed out that IR-based techniques can find information that is difficult for developers to find. Similar effects might occur in IR-based feature location techniques. In fact, some feature location techniques recommend relevant methods from an initial list, e.g. [26], [27]. These techniques are consistent with the behavior of the subjects observed in RQ3. They prevent developers from missing false negatives. If these techniques are used in the experiment, then manual validation might work better in adding false positives rather than removing false negatives.

Another internal validity is concern about the feature location tasks we designed. To avoid feature location tasks being overly dependent on a system, we choose feature requests from two systems. Additionally, we took two features from each system: one feature has a larger goldset; the other has smaller one, to avoid dependence to the size of the goldset. The task we prepared showed better and worse results for each of the four feature requests, and one task which gives the goldset itself. The result of our experiment might be dependent on those tasks. This threat can be decreased by adding more tasks and by replicating the experiment because we also make the dataset available online.

2) External Validity: As for the external validity, we believe that the results of our experiment can be generalized for use in other academic and industrial organizations. We recruited subjects from two universities and an industrial company and asked them to perform the same tasks. Because all of their organizations are specialized in software engineering, a weak threat exists by which the subjects might share some background in the discipline. We regard this threat as acceptable because their spectrum of programming experience ranges from a few years to 40 years.

We also believe that the result can be generalized to other systems written in Java language. However, the result might not be applicable in industry because the tasks are taken only from open-sourced systems.

3) Construct Validity: The major threat related to construct validity is that we used F-measure to assess the accuracy of both initial lists and validated lists. Although the F-measure captured the total improvement of precision and recall, different precision and recall values might result in the same F-measure. To avoid that problem, we analyzed precision and recall separately in Section V-D. However, a threat remains. If subjects often excluded a true positive from a list and included a false negative in the list, then the recall value is not changed. Such an effect is not readily apparent in the metric.

VII. Conclusion

As described in this paper, we have conducted a controlled experiment of feature location tasks. We have prepared lists of methods obtained using an automated technique, but their accuracy is controlled artificially. We asked 20 subjects to validate the lists manually. Consequently, the validated lists were totally improved from the initial lists. Developers could improve precision by recognizing false positives well, but they could not improve recall. This is true because a different understanding of a feature prevented the developers from recognizing true positives.

Several avenues of future effort remain. Due to reducing time effort for subjects, we kept the lists of methods small. How large a list of methods developers can reject false positives from the list should be investigated. Another important question is whether an incomplete feature location result is also effective for maintenance tasks, as similar to the effect of full requirements-to-code traceability reported by Mäder *et al.* [2]. Because developers understand more about a feature during their maintenance tasks, a partially located feature might be sufficient for developers. We are also interested in improving the feature location benchmark. We expect that the curation of goldsets described in Section IV can be automated to some extent by tracing changes on gold methods in the version history.

Acknowledgment

We would like to thank all the subjects who participated in this study. This work was supported by KAKENHI (Nos. 23680001 and 23700030).

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Softw.: Evol. and Proc.*, vol. 25, no. 1, pp. 53–95, 2013.

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

[3] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *Proc. ICSM*, 2011, pp. 213–222.

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

[5] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proc. WCRE*, 2004, pp. 214–223.

[6] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE TSE*, vol. 33, no. 6, pp. 420–432, 2007.

[7] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proc. ICPC*, 2008, pp. 53–62.

[8] D. Cuddeback, A. Dekhtyar, and J. H. Hayes, "Automated requirements traceability: The study of human analysts," in *Proc. RE*, 2010, pp. 231–240.

[9] W.-K. Kong, J. H. Hayes, A. Dekhtyar, and J. Holden, "How do we trace requirements? an initial study of analyst behavior in trace validation tasks," in *Proc. CHASE*, 2011, pp. 32–39.

[10] A. Dekhtyar, O. Dekhtyar, J. Holden, J. H. Hayes, D. Cuddeback, and W.-K. Kong, "On human analyst performance in assisted requirements tracing: Statistical analysis," in *Proc. RE*, 2011, pp. 111–120.

[11] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proc. ICSE*, 2012, pp. 430–440.

[12] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE TSE*, vol. 32, no. 12, pp. 971–987, 2006.

[13] M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" *JSS*, vol. 43, no. 1, pp. 19–27, 1998.

[14] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a static noninteractive approach to feature location," *ACM TOSEM*, vol. 15, no. 2, pp. 195–226, 2006.

[15] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *Proc. ICPC*, 2009, pp. 218–222.

[16] D. Cuddeback, A. Dekhtyar, J. H. Hayes, J. Holden, and W.-K. Kong, "Towards overcoming human analyst fallibility in the requirements tracing process," in *Proc. ICSE*, 2011, pp. 860–863.

[17] A. Ghabi and A. Egyed, "Code patterns for automatically validating requirements-to-code traces," in *Proc. ASE*, 2012, pp. 200–209.

[18] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. ISSTA*, 2011, pp. 199–209.

[19] D. Chatterji, J. C. Carver, B. Massengill, J. Oslin, and N. A. Kraft, "Measuring the efficacy of code clone information in a bug localization task: An empirical study," in *Proc. ESEM*, 2011, pp. 20–29.

[20] D. Binkley, D. Lawrie, and C. Uehlinger, "Vocabulary normalization improves IR-based concept location," in *Proc. ICSM*, 2012, pp. 588–591.

[21] M. Beard, N. Kraft, L. Etzkorn, and S. Lukins, "Measuring the accuracy of information retrieval based bug localization techniques," in *Proc. WCRE*, 2011, pp. 124–128.

[22] A. D. Eisenberg and K. D. Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *Proc. ICSM*, 2005, pp. 337–346.

[23] M. S. Christian Koppen, "PCDiff: Attacking the fragile pointcut problem," in *Proc. European Interactive Workshop on Aspects in Software*, 2004. [Online]. Available: http://pp.info.uni-karlsruhe.de/uploads/publikationen/stoerzer04eiwas.pdf

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, "Aspect oriented programming," in *Proc. ECOOP*, 1997, pp. 220–242.

[25] P. Mäder, O. Gotel, and I. Philippow, "Enabling automated traceability maintenance by recognizing development activities applied to models," in *Proc. ASE*, 2008, pp. 49–58.

[26] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *Proc. ESEC/FSE*, 2005, pp. 11–20.

[27] M. Trifu, "Improving the dataflow-based concern identification approach," in *Proc. CSMR*, 2009, pp. 109–118.

[28] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *Proc. ICPC*, 2012, pp. 193–202.