How developers perform feature location tasks: a human-centric and process-oriented exploratory study

Jinshui Wang1, Xin Peng1,*†, Zhenchang Xing2 and Wenyun Zhao1

1School of Computer Science, Fudan University, Shanghai, China
2School of Computing, National University of Singapore, Singapore

SUMMARY

Developers often have to locate the parts of source code that contribute to a specific feature during software maintenance tasks. This activity, referred to as feature location in software engineering, is a human-intensive and knowledge-intensive process. Researchers have investigated (semi-)automatic analysis-based techniques to assist developers in such feature location activities. However, little work has been carried out on better understanding how developers perform feature location tasks. In this paper, we report an exploratory study of feature location process, consisting of three experiments in which developers were given unfamiliar systems and asked to complete six feature location tasks. Our study suggests that feature location process can be understood hierarchically at three levels of granularity: phase, pattern, and action. Furthermore, our statistical analysis shows that these feature location phases, patterns, and actions can be effectively imparted to junior developers and consequently improve their performance on feature location tasks. Our qualitative observations and interviews also suggest that external factors, for example, human factors, task properties, and in-process feedbacks, affect the choices and usage of different feature location patterns and actions. Our results open up new opportunities to feature location research, which could lead to better tool support and more rigorous feature location process. Copyright © 2013 John Wiley & Sons, Ltd.

Received 13 February 2012; Revised 29 November 2012; Accepted 25 January 2013

KEY WORDS: feature location; human study; cognitive process; conceptual framework

1. INTRODUCTION

Developers often have to identify where and how a feature is implemented in source code in order to fix bugs, introduce new features, and adapt or enhance existing features. This activity is referred to as feature location [1] in the context of software engineering. In this paper, we use a broader meaning of feature location whose goal is to locate all the codes relevant to a given feature. This is driven by our earlier study [2] on feature location process, which shows that feature location is an iterative process involving not only the search of entrance points in the code for a feature but also the extension of entrance points to find more relevant code. Note that the search and extension phases correspond to feature location (a narrower meaning of feature location) and impact analysis, respectively, in existing literature [3].

Feature location has been recognized as one of the most common activities undertaken by software developers [4]. Because of the complexity of software systems and the cross-cutting concerns of features distributed in source code, the process of feature location is time-consuming and error-prone [5, 6]. To address this challenge, researchers have presented techniques to provide automated assistance in feature location tasks, using information retrieval (IR) [4, 7, 8], static analysis [8, 9], and dynamic

*Correspondence to: Xin Peng, School of Computer Science, Fudan University, Shanghai, China.
†E-mail: pengxin@fudan.edu.cn
This paper is a revised and extended version of a paper presented at ICSM 2011 in Williamsburg, USA.

Copyright © 2013 John Wiley & Sons, Ltd.
A comprehensive survey and taxonomy of feature location techniques can be found in [3]. Researchers have shown empirically that these proposed techniques can reduce the developer’s effort in locating the implementation of features and improve the quality of feature location results. Despite the success of these feature location techniques, feature location remains a human-intensive and knowledge-intensive activity, and it is risky to neglect human factors in the process [10]. Little research has been carried out on better understanding how developers perform feature location tasks. Thus, several important questions remain unanswered:

Q 1. What actions do developers commonly perform in the process of feature location?
Q 2. Are there recurring patterns that reflect different searching and extension strategies during feature location tasks?
Q 3. Are there distinct phases during feature location tasks? What are the purposes of these phases? How do developers start, proceed, and finish the task?
Q 4. Will the knowledge of feature location phases, patterns, and actions (if any) improve the performance of developers during feature location tasks?
Q 5. Are there external factors that may affect the developers’ choices and usage of feature location patterns and actions (if any), and how?

To begin to answer these questions, we have conducted an exploratory study of feature location process. This study consists of three experiments. The objective of the first experiment is to observe and analyze how senior developers accomplish feature location tasks (Q1, Q2, and Q3). More specifically, we recruited 20 developers from our graduate program and local companies. These developers were given two unfamiliar systems (JHotDraw and JPetStore) and asked to work on six feature location tasks in two 60-min sessions. We analyzed the course of their completion of the assigned tasks to identify elementary actions, recurring patterns, and distinct phases in the process of feature location.

The objective of the second experiment is to evaluate the effectiveness of the identified feature location phases, patterns, and actions in feature location tasks (Q4). We recruited 18 undergraduate students from our school. They were divided into two roughly ‘equivalent’ groups, based on their programming experience and capability. In the first 60-min session of this experiment, the participants of the two groups were asked to work on three feature location tasks on one of the two similar finance applications (JGnash and Buddi). Next, we introduced to the participants the feature location phases, patterns, and actions identified in the first experiment and then let the two groups swap the tasks. Finally, we comparatively investigated the performance of these participants (in terms of precision, recall, and F-measure [5] of their feature location results) during feature location tasks with and without the knowledge of feature location phases, patterns, and actions.

The objective of the third experiment is to examine the impact of self-learning on the quality of feature location results. This experiment is designed as a control experiment for the second experiment in order to strengthen our confidence in the effectiveness of the feature location knowledge on the performance of junior developers during feature location tasks.

In this experiment, we recruited 18 more undergraduate students from our school (different from those recruited in the second experiment). We used the same experiment settings as those of the second experiment, including subject systems, feature location tasks, development environments, and experiment sessions. The only difference from the second experiment is that in the third experiment, we no longer offered a tutorial session about the feature location phases, patterns, and actions between the two experiment sessions. That is, participants in the third experiment were not trained with feature location knowledge. However, these participants may still accumulate certain experience through self-learning in the whole experiment. This allows us to comparatively investigate the impact of self-learning on the performance of these 18 participants in the first and second sessions, respectively.

After the three experiments, we analyzed the screen-recorded videos of each participant’s feature location processes and conducted post-experiment questionnaires and interviews with the participants. The objective of such post-experiment analysis is to better understand the rationales and factors that affect the developers’ choices and usage of different feature location patterns and actions (Q5).
The contributions of the paper can be summarized as follows:

- We propose a hierarchical conceptual framework for understanding feature location process.
- We show that the explicit knowledge of feature location phases, patterns, and actions can improve the performance of the participants on feature location. Furthermore, it can also reduce the gaps between experienced developers and less experienced ones and the gaps between individual participants.
- Our analysis suggests that the improvement of the participants’ performance on feature location can be largely attributed to the participants’ ability to use feature location phases, patterns, and actions in a more complete and systematic way. We further confirm that the impact of self-learning on the performance of junior developers on feature location tasks is much less significant than that of the knowledge of feature location phases, patterns, and actions.
- Our analysis suggests that in addition to the knowledge of feature location phases, patterns, and actions, three external factors, that is, human factors, task properties, and in-process feedbacks, may also affect the choices and usage of feature location patterns and actions.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 discusses the design of our exploratory study. Section 4 reports the results and the analysis of the first experiment and proposes a conceptual framework for understanding feature location process. Section 5 reports the results and the analysis of the second and third experiments and discusses the impact of the knowledge of feature location phases, patterns, and actions on feature location process and quality. Section 6 discusses the impacts of external factors on feature location process. Section 7 summarizes the insights and lessons learned from our exploratory study. Section 8 discusses the external and internal threats to our study. Finally, Section 9 concludes and outlines our future plan.

2. RELATED WORK

Much of research on feature location (or traceability recovery in general) has been focused on (semi-)automatic techniques to alleviate the complexity and overwhelming information of software systems during feature location tasks and to improve the accuracy and quality of analysis results. In particular, researchers have investigated using IR [4, 7, 8], static analysis [8, 9], dynamic analysis [9], or a hybrid of several analysis techniques [4]. A systematic literature survey of feature location techniques has been presented in [3]. The usefulness and effectiveness of these techniques have been evaluated and demonstrated empirically [11, 12]. There are also a series of research focused on (semi-)automatic techniques for impact analysis. Researchers have investigated using static program slicing [13] or the combination of IR, dynamic analysis, and mining software repositories techniques [14, 15] to estimate an impact set given an initial set of program elements.

Despite the promising results of these (semi-)automatic feature location techniques, feature location remains a human-intensive and knowledge-intensive activity [16]. This gives rise to the need for a more detailed understanding of how developers perform feature location tasks. Egyed et al. [16] reported two exploratory experiments on recovering traceability links between requirements and code. Their work focused on the impact of task characteristics (e.g., system complexity and traceability granularity) and the effort and quality of recovering traceability links. Cuddeback et al. [17] presented a user study in which they investigated how human analysts examine candidate requirements traceability matrix produced by an automatic technique. However, little research has been carried out on better understanding how developers start, proceed, and finish the feature location tasks, what actions developers commonly perform, and what strategies (patterns) developers adopt in the process of feature location.

Raleigh [18] surveyed the field of program comprehension and provided a process model for concept location by dependency search as an example of general program comprehension process. Ko et al. [6] conducted an exploratory study on how developers understand unfamiliar code during software maintenance tasks. They found that developers interleaved three activities, that is, seeking, relating, and collecting relevant information. Furthermore, they argued the need for a general program understanding model based on these three activities and qualitatively discussed the implications of their findings for
software development tools. Sillito et al. [19] undertook two qualitative studies on how programmers ask and answer questions during programming change tasks. On the basis of an analysis of the data, they developed a catalog of 44 types of questions and categorized those questions into four categories, that is, finding focus points, expanding focus points, understanding a subgraph, and questions over groups of subgraphs.

In this work, we focused on one specific type of program understanding tasks, that is, feature location. Our study revealed a hierarchical conceptual framework for understanding the feature location process. At the highest level, this conceptual framework consists of four distinct phases: Seed Search, Extend, Validate, and Document. These phases are roughly at the similar level of abstraction to the three activities reported by Ko et al. [6]. We further investigated the actions directed toward the purpose of the four phases. We found out that the actions are not independent of each other. They form various patterns that reflect different strategies that developers adopt during feature location tasks. We quantitatively investigated the factors that affect the developers’ choices of different patterns and conducted a separate experiment to quantitatively analyze the usefulness and effectiveness of the identified feature location phases, patterns, and actions.

Demeyer et al. [20] described a set of useful patterns for reengineering object-oriented systems. Some of their reengineering patterns, for example, ‘Read all the Code in One Hour’ and ‘Step Through the Execution’, are similar to the elementary actions that developers commonly performed in our feature location study. However, our study presents a hierarchical conceptual framework for understanding feature location process, consisting of not only a set of elementary actions but also feature location phases with distinct purposes and recurring searching and extension patterns. Furthermore, our conceptual framework reveals that developers experience feature location as an interplay of feature location phases, patterns, and actions, whereas the reengineering patterns of Demeyer provide several guidelines for general program comprehension and reverse engineering.

This paper extends our earlier work [2] from the following four aspects. First, we further investigate Extend phase and present two recurring extension patterns in this paper. Second, to strengthen our confidence in the effectiveness of the feature location knowledge on the performance of junior developers, we conducted another experiment to examine the impact of self-learning on the quality of feature location results. Third, we further investigate the differences in the ways that developers perform feature location tasks before and after learning the feature location phases, patterns, and actions and how these differences contribute to the improvement of their feature location performance. Fourth, we examine the impacts of external factors (i.e., human factors, task properties, and in-process feedbacks) on the choices and usage of various feature location patterns and actions.

3. EXPERIMENT DESIGN

Our study consists of three separate experiments with distinct objectives, that is, the identification of distinct phases, recurring patterns, and elementary actions in feature location process; the evaluation of the effectiveness of the identified phases, patterns, and actions during feature location tasks; and the investigation of the impact of self-learning on the improvement of the quality of feature location results for the same subject systems and tasks. Table I summarizes the subject systems, the participants, and the number of feature location tasks of these three experiments.

3.1. Overview

Before each experiment, we introduced to the participants the background and relevant domain knowledge about the subject systems. Because we were interested in comparing the developers’ work on ‘identical’ tasks, we also explained the tasks to be completed and demonstrated the typical usage scenario(s) (if any) of the involved features, so that each developer would have a similar understanding of the assigned tasks. During each experiment, we had two assistants who were only allowed to answer clarification questions about the task descriptions or assist the participants in configuring the development environment. The assistants were also responsible for ensuring that the participants would not discuss or share their solutions.
In the first experiment, all 20 participants were given three feature location tasks on JHotDraw and three tasks on JPetStore. They had a 60-min session to complete as many tasks and accurately as possible for each subject system. They had a 10-min break between the two sessions.

In the second experiment, all 18 participants were divided into two equivalent groups ($T_1$ and $T_2$), based on their programming experience and capability. In the first 60-min session (before session), the participants of group $T_1$ were given three feature location tasks on JGnash, whereas those of $T_2$ were given three tasks on Buddi. After the first session, we gave a tutorial about the feature location phases, patterns, and actions identified in the first experiment. We explained the main phases during feature location tasks and their purposes, the benefits and pitfalls of different search patterns in an objective manner without bias, and the actions that participants may perform and the types of information that they may seek. We also answered the questions raised by the participants. After a 10-min break, the two groups swapped the subject systems and completed the other three tasks in the second 60-min session (after session).

In the third experiment, we used the same experiment settings as those of the second experiment. That is, we also divided the 18 participants into two roughly equivalent groups ($T_3$ and $T_4$). The two groups were asked to work on three feature location tasks on JGnash and Buddi, respectively, in the first session, and then, they swapped the subject systems and their tasks in the second session. The only difference from the second experiment is that in the third experiment, we no longer offered a tutorial session about the feature location phases, patterns, and actions between the two experiment sessions.

The participants in all three experiments were asked to fill in a post-experiment questionnaire to rate (based on a 1 to 5 scale) the perceived difficulty of their tasks, whether they had enough time to perform the tasks. The participants in the second experiment were also asked to rate the perceived usefulness of the feature location knowledge they were taught and which phase was the most difficult and why.

Because we need to analyze the actions and processes of each participant in completing the assigned tasks, we required the participants to run a full-screen recorder once they started working on the tasks. Furthermore, we encouraged participants to record by audio (i.e., think-aloud protocol) the important moments during their feature location process, such as when they began, finished, or switched to which tasks. However, we did not enforce the think-aloud protocol for every actions that those participants may perform. Although such information may facilitate our post-experiment analysis, we believe that such think-aloud protocol would significantly interfere with the participants’ normal feature location processes. Such interference is called Hawthorne effect [21]. For example, participants may have to divert from their feature location tasks in order to phrase how to properly express their actions and intentions.

<table>
<thead>
<tr>
<th>Experiment 1</th>
<th>Experiments 2 and 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>JPetStore</td>
</tr>
<tr>
<td>System type</td>
<td>Web application</td>
</tr>
<tr>
<td>Tasks</td>
<td>3</td>
</tr>
<tr>
<td>Time (min)</td>
<td>60</td>
</tr>
<tr>
<td>Version</td>
<td>7.4.1</td>
</tr>
<tr>
<td>Classes</td>
<td>700</td>
</tr>
<tr>
<td>Methods</td>
<td>7256</td>
</tr>
<tr>
<td>LOC</td>
<td>72,911</td>
</tr>
<tr>
<td>Participants</td>
<td>18 graduate students and 2 full-time developers</td>
</tr>
<tr>
<td>Goal</td>
<td>Observe and analyze phases, patterns, and actions in feature location process</td>
</tr>
<tr>
<td>JGnash</td>
<td>Buddi</td>
</tr>
<tr>
<td>System type</td>
<td>Finance application</td>
</tr>
<tr>
<td>Tasks</td>
<td>3</td>
</tr>
<tr>
<td>Time (min)</td>
<td>60</td>
</tr>
<tr>
<td>Version</td>
<td>2.6.0</td>
</tr>
<tr>
<td>Classes</td>
<td>511</td>
</tr>
<tr>
<td>Methods</td>
<td>3975</td>
</tr>
<tr>
<td>LOC</td>
<td>43,957</td>
</tr>
<tr>
<td>Participants</td>
<td>Experiment 2: 18 third-year and fourth-year undergraduate students</td>
</tr>
<tr>
<td>Goal</td>
<td>Experiment 2: evaluate the effectiveness of the identified feature location knowledge</td>
</tr>
<tr>
<td></td>
<td>Experiment 3: 18 third-year and fourth-year undergraduate students</td>
</tr>
<tr>
<td></td>
<td>Experiment 3: examine the impact of self-learning</td>
</tr>
</tbody>
</table>

LOC, lines of code.
The participants were asked to document their feature location results according to the given template, and they submitted their results at the end of each experiment. The feature location results include a mandatory list of methods for each task and an optional brief description for each method. We evaluated the feature location results of each participant with $F$-measure, that is, the weighted harmonic mean of recall and precision [5]. A small incentive was offered to all the participants, and the top three participants who provided the best results were offered an extra box of candy.

3.2. Participants

As our study consists of three separate experiments with distinct objectives, we recruited three entirely different groups of participants in the three experiments.

In the first experiment, we recruited 20 developers, including 18 senior graduate students from our graduate program and two full-time developers from local companies. Two student participants worked as professional developers in industry before they entered our graduate program; six students participated in the development of industrial projects (e.g., internships) during their graduate study; the remaining 10 had at least 1 year of experience on the design and development of various research tools. The two full-time developers had on average 5 years of industrial experience. On the basis of our pre-experiment survey, all 20 developers described themselves as ‘Java experts’ and had rich experience with desktop applications, Web-based applications, or both. All 20 developers used Eclipse in their daily work. Seven out of 20 participants reported that they had experiences with design patterns in general, but none of the participants had experiences with the two subject systems JHotDraw and JPetStore.

In the second experiment, we recruited 18 third-year and fourth-year undergraduate students from our school. On the basis of our pre-experiment survey, all of them used Eclipse ‘regularly’ and reported on average 9.64 h of programming a week. A total of 14 students described themselves as ‘above-average’ Java experts, and the remaining four described themselves as ‘Java experts’. Before this experiment, nine students only had experience with small projects less than 2000 lines of code (LOC); six students had experience with projects of 2000 to 10,000 LOC, and the remaining three had experience with projects of over 10,000 LOC. We surveyed the capability of the participants from several aspects, including software development/maintenance experience, familiarity with Java, and familiarity with Eclipse. For each aspect, a score ranging from 1 to 5 (1 being the lowest and 5 being the highest) was given. On the basis of this survey, we obtained the overall capability score for each participant by computing the average of his scores. We then divided the 18 participants into two equivalent groups based on their capability scores (Table II). This allowed us to perform a ‘fair’ comparison of their performance on the same set of tasks.

In the third experiment, we recruited 18 more third-year and fourth-year undergraduate students from our school (different from those of the second experiments). On the basis of our pre-experiment survey, 16 of them used Eclipse regularly, and the remaining two use Eclipse ‘sometimes’, and all reported on average 9.13 h of programming a week. Eleven students described themselves as ‘above-average’ Java experts, five students described themselves as ‘below-average’ Java experts, and the remaining two described themselves as ‘Java experts’. Before this experiment, eight students only had experience with small projects less than 2000 LOC; seven students had experience with projects of 2000 to 10,000 LOC, and the remaining three had experience with projects of over 10,000 LOC. Similar to the second experiment, these 18 students were also divided into two roughly equivalent groups based on their capability scores (Table II).

<table>
<thead>
<tr>
<th>CapabilityScore</th>
<th>Experiment 2</th>
<th>Experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\geq 4$</td>
<td>$T_1$ 1</td>
<td>$T_3$ 1</td>
</tr>
<tr>
<td>$\geq 3$</td>
<td>$T_2$ 3</td>
<td>$T_4$ 0</td>
</tr>
<tr>
<td>$\geq 2$</td>
<td>$T_3$ 4</td>
<td>$T_4$ 4</td>
</tr>
<tr>
<td>$&lt; 2$</td>
<td>$T_4$ 2</td>
<td></td>
</tr>
</tbody>
</table>

3.3. Subject systems

Our study involves four open-source Java systems as summarized in Table I. The selection of these four subject systems was driven by the objectives of our experiments.

In the first experiment, we used JHotDraw and JPetStore, because we were interested in finding a variety of feature location strategies in systems of different size and nature. This allowed us to obtain more comprehensive observations on how senior developers accomplish feature location tasks:

- JHotDraw (http://www.jhotdraw.org) is a Java GUI framework for technical and structured graphics. It supports a default diagram editor with basic editing features. The JHotDraw 7.4.1 used in this study consists of 700 classes and 72.9K lines of Java code.
- JPetStore (https://src.springframework.org/svn/spring-samples/jpetstore) is a demonstration application for the Spring framework, adapted from the original PetStore. The JPetStore 1.0.0 used in this study consists of 73 classes and 2.3K lines of Java code.

JHotDraw is the largest subject systems used in our study, whereas JPetStore is the smallest. Their sizes are different in one order of magnitude. JHotDraw is a desktop application. Its design relies heavily on design patterns [22]. Design patterns introduce delegations to the implementation, which may hinder the exploration of related program elements. Furthermore, ‘programming to interface’ tenet followed by design patterns may affect the search for concrete implementation in a class hierarchy. JPetStore is a Web-based application, built on the Spring framework. The Spring framework relies heavily on dependency injection. This may increase the difficulty to feature location, because one has to understand how the framework hooks up different components. Finally, most of the features of JHotDraw are directly or indirectly related to GUI, whereas most of the features of JPetStore are related to database operations.

In the second experiment, we were interested in the comparative evaluation of the developers’ performance on feature location tasks. Thus, we used JGnash and Buddi, which are two systems of comparable size and from the same domain (personal finance):

- JGnash (http://freshmeat.net/projects/jgnash) is a personal finance application. It supports several account types, nested accounts, scheduled transactions, and currencies. The JGnash 2.6.0 used in this study consists of 511 classes and about 44K lines of Java code.
- Buddi (http://buddi.digitalcave.ca) is a simple budgeting application targeted for users with little or no financial background. It allows users to set up accounts and categories, record transactions, check spending habits, and others. The Buddi 3.4.0 used in this study consists of 253 classes and about 18.8K lines of Java code.

Although the two subject systems are from the same domain, they do not share any common features. We intentionally did so to avoid the bias that may be incurred in the second session of the second experiment by the domain knowledge that participants accumulated in the first session. Furthermore, we carefully designed feature locations tasks so that participants have to explore different aspects of similar features of the two subject systems, such as different data access mechanisms.

In the third experiment, we were interested in examining the impact of self-learning on the quality of feature location results. Thus, we used the same subject systems as those of the second experiment, that is, JGnash and Buddi.

3.4. Tasks

In our study, each task is concerned with one specific feature either relevant to a UI element, a database operation, or an algorithm. Each participant was given a sheet of paper describing their feature location tasks on a given subject system. Each task came with a short feature name, a free-form textual description, and at least one typical usage scenario (Table III). For each task, the participants were requested to identify as many methods that they deemed to be relevant to the given feature as possible. Developers had freedom to complete the assigned tasks in any order they preferred. They were also allowed to switch to another task if they found relevant information to that task while they were in the middle of a task. They were instructed to record (by audio) the important moments, such as when they begin, finish, or switch to which tasks.
We designed three categories of feature location tasks, including four UI-related tasks, three program-
internal tasks, and five database-related tasks. This design allowed us to investigate variations in the
developer’s strategies on different nature of tasks. Table III shows an example of each category.

<table>
<thead>
<tr>
<th>Feature Scenario</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Group/Ungroup Graphic Elements (JHotDraw)</td>
<td>Select several target elements on the canvas → Right click → Select Group operation in the context menu</td>
</tr>
<tr>
<td>Add New Transaction (JGnash)</td>
<td>Select an account within Account List window → Fill in translation data → Click Enter button</td>
</tr>
<tr>
<td>Auto Save (Buddi)</td>
<td>Select Edit Menu → Preferences → Advanced tab → Set the value of Auto Save Period → Click Ok button</td>
</tr>
</tbody>
</table>

Table III. Task examples.

We designed three categories of feature location tasks, including four UI-related tasks, three program-
internal tasks, and five database-related tasks. This design allowed us to investigate variations in the
developer’s strategies on different nature of tasks. Table III shows an example of each category.

‘Group/Ungroup Graphics Element’ is a UI-related task for JHotDraw. It requires developers to make
good use of various UI-related hints, such as tooltips and graphic widgets to explore the system. ‘Add
New Transaction’ is a database-related task for JGnash. It requires developers to effectively explore
project packages and static dependencies relevant to database operations. ‘Auto Save’ is a program-
internal task for Buddi. This feature requires developers to investigate the other relevant feature
‘preferences management’ in order to find the code relevant to Auto Save.

3.5. Development environment

The participants were given the Eclipse 3.4.2 environment and the source code of the subject systems
loaded as Eclipse project. They were allowed to use debuggers, text editors, and paper for notes. They
were also allowed to search the Internet for additional information about the subject systems. Before
the experiments, we browsed the internet to ensure that there exist no ‘sweet’ answers to their
assigned tasks. However, they were not allowed to download and use existing feature location tools,
such as FLAT3 [23] or Re-Trace [11], because the primary objectives in this study are not to
evaluate the effectiveness of such feature location tools.

3.6. Measures

We evaluated the quality of the participants’ feature location results (i.e., their performance) using
F-measure. To avoid participants playing tricks, we did not inform them what measure we would
adopt to evaluate their results.

Let T be a feature location task consisting of a set of features F. Let M be the set of methods in the
implementation. Given a feature f ∈ F, let \( L^f_{\text{actual}} \) be the set of actual traceability links between
the feature f and the methods M. In our study, we together with two experts who are familiar with the
subject systems manually located the methods for all the features involved in our feature location
tasks. This provides the ground truth* (i.e., \( L^f_{\text{actual}} \)) for evaluating the participants’
results.

Given \( f \in F \), let \( L^f_{\text{reported}} \) be the set of traceability links between the feature f and the methods M,
reported by a participant. Precision \( P_f \) is the percentage of correctly reported traceability links, that is,
\( (L^f_{\text{reported}} \cap L^f_{\text{actual}}) / L^f_{\text{reported}} \). Recall \( R_f \) is the percentage of actual traceability links reported, that is,
\( (L^f_{\text{reported}} \cap L^f_{\text{actual}}) / L^f_{\text{actual}} \). Given a feature location task consisting of a set of features F, we compute
the overall precision \( P_T \) and recall \( R_T \) for the task T as follows: \( P_T = \sum_{f \in F} P_f / |F| \), \( R_T = \sum_{f \in F} R_f / |F| \).

Achieving good precision and good recall is a balancing act (as precision increases, recall tends to
decrease and vice versa), and F-measure represents a balancing metric, indicating the combination of
precision and recall [5]. In our study, we found that the participants usually returned more results in
their after sessions after learning the feature location knowledge. This often caused a significant
increase on recall with a slight decrease on precision. Therefore, we use F-measure to measure the
participants’ overall performance on feature location tasks, which can be interpreted as a weighted
average of the precision and recall. F-measure for a feature location task T is then computed as

\[ F = \frac{2 \times P_T \times R_T}{P_T + R_T} \]

*The ground truth is available at http://www.se.fudan.edu.cn/research/jsme2012featurelocation/groundtruth.pdf
In this work, following the treatment in [5], we set $b$ to 2, that is, recall is considered four times as important as precision, because we deem that finding missing traceability links is more difficult than removing incorrect links.

3.7. Data analysis

Our experiments produced 112 h of full-screen videos of 56 developers’ work on 12 feature location tasks on four subject systems. Our analysis follows grounded theory [24] commonly used in interaction design to structure the analysis of observational data gathered in the user study. Grounded theory is an approach to qualitative data analysis that aims to develop theory from the systematic analysis and interpretation of empirical data, that is, the theory derived is grounded in the data. Our analysis involves iterative categorization of data, identification of recurring patterns and critical events, and refinement of analysis results [24].

After the experiments, we analyzed these videos as follows. First, we watched each developer’s video to find the moments when the developer began, finished, or switched the tasks. A total of 63% of developers followed our instruction to record (by audio) some explanation (maybe incomplete) for such important moments. Furthermore, we looked for other cues, such as reading task descriptions and closing all or most of the opened files. Once we determined the sequence of tasks that a developer worked on, we examined each task in detail, observing the developer’s actions and noting any interesting patterns and phases regarding how developers perform feature location tasks.

The analysis of 112 h of screen-recorded videos requires a few hundred hours of manual work, which is tedious and error-prone. Inspired by the key principle of pair programming, in our analysis, we let two authors cooperatively examine the screen-recorded videos. This pair inspection allows us to achieve improved discipline and time management, less likely skip or miss important information, and keep both inspectors honest. Furthermore, the pair inspection allows the two inspectors to identify the discrepancies in their interpretation and reach consensus as early as possible in a cooperative manner. This reduces the risk of having to adjust their interpretation or even redo the manual inspection again in the later stage of analysis.

To assist our analysis, we developed a simple tool called LogHelper (Figure 1). This tool allowed us to create and maintain a log of each developer’s work as we analyzed his task videos. LogHelper allows us to log important information such as the developer’s actions, their start and stop time, and the order of these actions. It can also log our comments and inferences about the developers’ actions, such as the keyword they used in search, the types of dependencies they explored, the APIs they inspected, and the location

Figure 1. The tool LogHelper for pattern analysis.
of breakpoints. With the support of LogHelper, we essentially transcribed the screen-recorded video of the developers’ feature location tasks into a transcript of the developers’ actions during the tasks.

For example, during the second experiment, one developer showed the following behaviors when he was doing the ‘Edit Exchange Rate’ task on JGnash. He first read the scenario description of the task and executed the program. During the program execution, he stayed on the main UI of the system and moved his mouse pointer to some tooltips on the UI. Then, he searched in source code with the keywords ‘edit exchange rates’ and got no results returned. After that, he searched again with the new keyword ‘exchange’ and obtained 630 results. After checking the returned results, he changed to use Java search and had 43 results by searching in method declarations with the pattern ‘*exchange*’. He then checked some results by opening the selected methods in an editor and reading the code. After several trials, he began to toggle breakpoints in one selected method.

On the basis of the earlier behaviors observed from the task video, we inferred a series of physical actions, including review task description, run program, search program elements, inspect search result, and read code. Some original actions were generalized or merged to more general actions. For example, Java search and file search were generalized to search program element, and read source code and read Javadoc were merged to read code. On the basis of these observable actions, we inferred some mental actions. For example, after checking the results, the developer changed his keywords and did code search again, and accordingly, we inferred that he had a mental action ‘enrich/refine keywords’ during that period.

Given the transcripts of the developers’ actions during feature location tasks, we then conducted a bottom-up analysis to derive recurring patterns. We first identify frequent sequences of actions and then gradually combined shorter frequent sequences into longer ones. During this process, similar frequent sequences may be merged, and their differences will be modeled using branches and/or optional actions. Different types of patterns emerge from recurring sequences. For example, on the basis of the transcript obtained from the task video discussed earlier, we can identify some frequent sequences, such as ‘search program elements, inspect search result, and read code’. This sequence of actions occurred several times in the task video. The different behaviors of the developer when he was faced with a large number of search results or a small number of search results can be merged as a branch depending on the number of returned results (i.e. too many results versus a few results). By combining recurring sequences of actions and branching(optional actions), we can derive the IR-based search pattern as shown in Figure 3.

Finally, we determine the distinct purposes of the derived patterns and group patterns into phases serving different purposes. For example, when we observed that the developer started debugging the code, we inferred that he finished his Seed Search phase and entered the Extend because he seemed to find a good entrance point and started extending from it.

4. FEATURE LOCATION PHASES, PATTERNS, AND ACTIONS

We now describe and assess both quantitatively and qualitatively the empirical results that we collected in our study and the lessons and insights we learned from our empirical results. In this section, let us first review the empirical results of the first experiment (Sections 4.1, 4.2, and 4.3). We then propose a hierarchical conceptual framework for understanding feature location process based on the empirical results of the first experiment (Section 4.4).

4.1. Actions

In the first experiment, we observed 17 initial types of physical actions and 16,203 physical actions in total. We removed three types of uncommon actions, that is, actions used less than 15 times, such as six ‘reading irrelevant non-source files’ actions. These removed actions have little influence on feature location processes. Furthermore, we merged several relevant types of actions. For example, we initially had a ‘switching source files’ action. Further analysis revealed that developers usually switched between a set of relevant source files that they opened earlier through Open Declaration or other Eclipse’s navigation mechanisms. Thus, we classified the ‘switching source files’ actions as
exploring static dependency’ actions. Such merging of relevant types of actions abstracts certain
details of actual actions that developers perform during feature location process. This makes it easier
to analyze and understand the feature location process. Finally, we obtained 11 types of physical
actions that developers commonly performed during feature location tasks in the first experiment.
These are listed in Table IV. In the first experiment, developers performed a median of 213 (± 49)
actions per task.

As shown in Table IV, some types of physical actions were used more frequently than others. This is
unsurprising, because actions such as search program elements, explore static dependency, and
breakpoint operations/step program represent three basic techniques to understand software, through
textual, static, or dynamic information. In contrast, actions, such as review task description and
document results, represent specific-purpose actions in the feature location process. For example,
document results are used to record the program elements that developers deem to be relevant to the
given feature. Although developers may document results several times, it has been used much less
frequently than general-purpose actions, such as reading code.

In addition to 11 types of physical actions, we also inferred six types of mental actions, including
conceive an execution scenario, derive a variant scenario, identify relevant keywords, enrich/refine
keywords, hypothesize relevant files, and hypothesize a code position. These actions represent some
analysis that developers likely perform in mind. The inference of mental actions is based on the
assumption that the observable behaviors of developers are led by their rational thoughts, that is,
mental actions. The principle of the inference of mental actions can be summarized as follows: a
mental action takes information from previous actions as input and produces guidelines for the
following actions as output. For example, an instance of enrich/refine keywords can be inferred from
the facts that search results returned by previous searches provide necessary feedback for the
adoption and refinement of keywords in a sequence of searches.

Although this list of mental actions is by no means complete, it allows us to better understand the
developers’ behavior during feature location tasks. Note that these mental actions may or may not
have physical indicators. We inferred them from screen-captured videos and post-experiment
interviews. For example, we looked for cues in the videos, such as pauses in activity after the
developer performed search, exploration or execution actions, the repetitive highlighting of a code
fragment, or hovering over a program element. Although these cues were not without uncertainty,
they allowed us to approximate the period that participants may perform some mental analysis. Then,
we consulted with participants in the post-experiment interview about their activities in such
periods.

4.2. Phases

By studying logs of the developers’ actions, we observed four interleaved phases that fulfill four
distinct purposes, that is, Seed Search, Extend, Validate, and Document. Figure 2 summarizes these
four phases and their purposes in the feature location process. Table V summarizes the heuristics

<table>
<thead>
<tr>
<th>Type</th>
<th>Information or activity</th>
<th>Frequency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read code</td>
<td>Source code, comments, Javadoc, API specification</td>
<td>27.2</td>
</tr>
<tr>
<td>Search program elements</td>
<td>Search text string or Java element, inspect search result</td>
<td>8.5</td>
</tr>
<tr>
<td>Explore static dependency</td>
<td>Type hierarchy, declaration, reference (call, access)</td>
<td>21.1</td>
</tr>
<tr>
<td>Explore packages</td>
<td>Package Explorer view, outline view</td>
<td>11.3</td>
</tr>
<tr>
<td>Breakpoint operations</td>
<td>Toggle breakpoints, disable/enable breakpoints</td>
<td>13.2</td>
</tr>
<tr>
<td>Step program</td>
<td>Step into/out, step over, suspend</td>
<td>6.6</td>
</tr>
<tr>
<td>Document results</td>
<td>Bookmark, copy/paste to a document, writing notes</td>
<td>3.8</td>
</tr>
<tr>
<td>Review task description</td>
<td>Task description</td>
<td>2.8</td>
</tr>
<tr>
<td>Run program</td>
<td>Execute without breakpoints</td>
<td>2.8</td>
</tr>
<tr>
<td>Edit code</td>
<td>Add/remove comments, print out messages</td>
<td>1.4</td>
</tr>
<tr>
<td>Browse external resources</td>
<td>Internet; dictionary</td>
<td>1.4</td>
</tr>
</tbody>
</table>
that we examined to identify the phases of the feature location process, mainly depending on the typical actions involved in different phases. It is important to note that developers may perform the same type of actions for completely different purposes. For example, instead of extending the relevant program elements, a developer may explore the static dependencies to validate the intent and behavior of an element. In our study, we combined the heuristics listed in Table V and other contextual information to determine the phases of the developers’ work during feature location tasks.

Because of the nature of feature location tasks, developers in our study always began their feature location processes by searching for entrance points, that is, a minimal set of program elements to start their exploration. They used various techniques and explored different kinds of information. Some began with a textual search for what they perceived to be relevant keywords, based on, for example, feature names or task descriptions. They may also test run the program and observe cues for keywords. Some began by scanning the packages and files in the Eclipse Package Explorer view and further reading/exploring the files that they deemed relevant. Others began by conceiving some execution scenarios and debugging the program.

Once developers determined a set of entrance points, they usually attempted to extend this set to find more relevant program elements. Some developers followed static dependencies, such as type hierarchy, declaration, reference (e.g., method call and field access). Some used the Eclipse’s code highlighting feature to quickly scan the relevant program elements within a source file. Others relied on programming debugging (especially stepping into) to explore the relevant program elements on the execution traces from an entrance point. A quantitative analysis about how often different searching/extending strategies (described as patterns, see Section 4.3) were used during feature location tasks and their potential impact on the quality of feature location results can be found in Section 5.2.

---

Figure 2. Four distinct phases of feature location process.

Table V. Heuristics for phase identification.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Identification heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed Search</td>
<td>The start of a new task&lt;br&gt;After documenting-results actions&lt;br&gt;Frequent search of program elements&lt;br&gt;Frequent static-dependency exploration&lt;br&gt;Stepping into the program&lt;br&gt;Run the program and observe&lt;br&gt;Browsing external resources</td>
</tr>
<tr>
<td>Extend</td>
<td>Return to an element and explore its other information&lt;br&gt;Frequent static-dependency exploration&lt;br&gt;Stepping into the program</td>
</tr>
<tr>
<td>Validate</td>
<td>Toggling/Enabling breakpoints&lt;br&gt;Quickly stepping over the program&lt;br&gt;Editing code and run the program&lt;br&gt;Printing out messages</td>
</tr>
<tr>
<td>Document</td>
<td>Bookmarks&lt;br&gt;Copy/Paste elements to a document&lt;br&gt;Writing notes</td>
</tr>
</tbody>
</table>
Note that it was sometimes difficult to clearly determine the boundary between the Seed Search and Extend phases, because these two phases tended to be interleaved. Furthermore, developers may perform similar actions during these two phases, such as exploring static dependency or stepping the program. Our observation was that the developers’ program exploration and execution tended to be more general without very clear targets during the Seed Search phase, whereas their actions were more specific during the Extend phase. For example, during the Extend phase, they often kept returning to a relevant element and explored its other dependencies.

After identifying some relevant program elements, developers often attempted to determine the relevance of these program elements to the given feature. Some developers edited the code (e.g., print out certain messages) and executed the program. Others relied on programming debugging again. In contrast to the Seed Search and Extend phases, developers in the Validate phase tended to quickly step over the program to the point they wanted to inspect.

In the last Document phase, developers documented the program elements that they deem relevant by bookmarking these elements, copying them into a document, or writing notes. Because of the limitations of integrated development environments and tools, in our experiments, developers only simply recorded the names of program elements relevant to the given feature. However, we believe more detailed and useful information such as keywords for search and rationales behind dependency exploration should also be recorded in this phase. Such information characterizes the process and rationale of successful feature location processes and can provide useful knowledge for future program understanding and evolution tasks.

Finally, if they believed that there are no more relevant program elements, they finished the task at hand. Otherwise they started a new round of Seed Search, Extend, and Validate activities. Usually, we can use the following two heuristics to detect the ending of a task: the developer records the results and then begins to read the description of the next task; and the developer begins another task by searching with different keywords or scenarios. For the latter case, as the keywords and scenarios for different tasks are quite different, it is easy to differentiate between starting a new task and continuing the current task.

It is important to note that the four phases illustrated in Figure 2 represent a reference process for understanding feature location activities. There often exist no clear boundaries between the four phases. Furthermore, developers may not perform all the four phases during feature location tasks. For example, after identifying a set of entrance points, a developer may jump to the Validate phase without going through the Extend phase. As another example, if a developer believes that the program elements being examined are not relevant, he may go back directly from the Validate phase to the Seed Search phase, without going through the Document phase.

4.3. Patterns

We now review the recurring search and extension patterns that we observed in the feature location processes of the participants in the first experiment.

4.3.1. Search patterns. On the basis of the identified phases of the developers’ work, we analyzed how much time each developer in the first experiment spent on different phases. Overall, developers spent about half of their time searching for entrance points, forth of their time extending the relevant program elements, sixth of their time validating the relevance of program elements, and twelfth of their time documenting their analysis results.

This division of the developers’ time on different phases is a rather rough estimation. First, we allowed developers to switch to another task while they were in the middle of a task. Second, phases sometimes cannot be clearly separated, for example, the interleaving short Seed Search and Extend phases. Third, we conducted experiments in 60-min sessions. Developers often tried to improve their feature location results toward the end of the experiments, if they had spare time after finishing all three tasks. It is difficult to classify their work in this period of time. However, our analysis still suggested that developers spent much more time on Seed Search phase than the other three phases.

Furthermore, the post-experiment questionnaires returned by the participants suggested that they consider finding the entrance points as fast and accurately as possible as the most important activity during feature location tasks. Our post-experiment analysis also suggested that the success of searching for entrance points has the biggest impact on the quality of feature location results, because it involves many speculation, errors, and useless operations.
The importance of Seed Search phase and the challenges that developers may face in this phase motivated us to further study the developers’ actions within the Seed Search phase. We identified three search patterns in the developers’ work in the first experiment that reflect different strategies that developers adopted to find entrance points in the Seed Search phase.

4.3.1.1. Information retrieval-based search pattern. The IR-based search pattern (Figure 3) finds the entrance points for a feature location task by IR techniques. A developer who adopts this pattern begins by identifying textual keywords that he perceives to be relevant to the feature. He then searches program elements using these keywords (e.g., by Eclipse Find or Search File/Java). The initial search often returns many results. The developer usually refines his search from cues he observes in the search results. Finally, he reads the code to determine whether the research results contain the potential entrance points.

A developer usually tries to first identify keywords from feature descriptions. However, when he fails to find relevant entrance points based on feature descriptions, he may optionally conceive an execution scenario and run the program to identify more cues for keywords. A developer may even begin his search by running the program, especially when the feature is UI-related. This allows him to identify keywords from UI (e.g., tooltip and graphic widget) and program output. When the search fails to find relevant entrance points, the developer may conceive a different scenario or revise the current scenario to find more cues.

4.3.1.2. Execution-based search pattern. Execution-based search pattern (Figure 4) finds the entrance points by conceiving some execution scenarios and stepping the program. Note that ‘toggle breakpoint’ is an instance of ‘breakpoint operations’; however, considering that toggle breakpoint is more meaningful in this context, we use toggle breakpoint instead of breakpoint operations in Figure 4.
A developer who adopts this pattern begins by conceiving an execution scenario that he deems to be relevant to the feature. He then quickly explores packages to identify a few source files that seem potentially relevant. Next, he attempts to set some breakpoints to debug the program. The initial breakpoints are usually very imprecise, such as in main method or all action listeners. The developer gradually adjusts breakpoints on the basis of program execution results, for example, whether the breakpoints are reached, whether the potentially relevant source files are exhaustively explored, or whether a different or variant scenario needs to be considered. Once a breakpoint is reached, the developer steps the program and reads the relevant code. He may run and debug the program a few times to inspect different execution paths or different execution scenarios.

It is important to note that the roles that program execution plays in IR-based and execution-based search patterns are completely different. The program execution in IR-based search pattern provides an alternative way to identify keywords for program search. That is, it plays a supportive role to the primary technique, that is, search program elements. Therefore, developers usually run the program without breakpoints and stepping in IR-based search pattern. In contrast, program execution is the primary technique used in execution-based search pattern to find entrance points. Developers rely heavily on breakpoint operations and program stepping.

4.3.1.3. Exploration-based search pattern. Exploration-based search pattern (Figure 5) finds the entrance points by exploring static program dependencies. A developer who adopts this pattern begins by exploring packages and hypothesizes for a set of potentially relevant source files. In contrast to the package exploration in execution-based search pattern, the developer here tends to expand more files (in Eclipse Package Explorer view) to inspect the program elements defined in the files or open more files to read their code. Furthermore, the developer follows static program dependencies (e.g., method calls, field access, and type hierarchy) to reach more potentially relevant program elements.

4.3.2. Extension patterns. The post-experiment questionnaires returned by the participants suggested that Extension phase was considered as another important activity during feature location tasks, especially for achieving high recall in feature location tasks. In fact, our post-experiment analysis suggested that the low recall of the feature location results of the participants was usually due to the participants’ inability to effectively accomplish Extension phase in order to locate more relevant program elements based on seed entrance points. Our further analysis of the developers’ actions within the Extension phase identified two extension patterns that reflect different strategies that developers adopted to find more relevant program elements in the Extend phase.

4.3.2.1. Execution-based extension pattern. Similar to execution-based search pattern, execution-based extension pattern (Figure 6) is also based on conceiving some execution scenarios and stepping the program. Execution-based extension pattern can be regarded as a variant of execution-based search pattern.

However, there are some essential differences between them. First, their purposes are different. The purpose of execution-based search pattern is to find the entrance points of a given feature, whereas execution-based extension pattern is to explore from entrance points to identify more relevant program elements. Their different purposes in turn result in the differences in how developers conceive and use execution scenarios.

For execution-based search pattern, developers focus on choosing suitable execution scenarios that can accurately reveal appropriate entrance points for the implementation of the given feature. In contrast, for execution-based extension pattern, the challenge is to achieve a good coverage of
potentially relevant program elements, that is, trying to identify as many relevant program elements of the given feature as possible. Therefore, developers focus on executing sufficient scenarios starting from the chosen entrance points and stepping relevant execution paths. As a result, the developer does not need to hypothesize relevant files in execution-based extension patterns. As the developer knows the context of a selected entrance point, he is able to hypothesize some code positions for setting breakpoints directly. Furthermore, the developer may run and step the program several times to examine different execution scenarios and different execution paths to discover as many relevant program elements of the given feature as possible.

Note that some developers may document the potentially relevant program elements as they step the program and read code, whereas others may remember such potentially relevant program element in mind first and then document them after validating their relevance. The optional action 'collect relevant elements' in execution-based extension pattern represents both types of actions that developers may perform.

4.3.2.2. Exploration-based extension pattern. Similar to exploration-based search pattern, exploration-based extension pattern (Figure 7) identifies relevant program elements by exploring static program dependencies. Exploration-based extension pattern thus can be regarded as a variant of exploration-based search pattern.

The key difference lies in the fact that by using exploration-based extension pattern, a developer frequently returns to focus point (i.e., a seed entrance point or a program element reachable from a seed entrance point that is used as the starting point of the subsequent extensions) and explores its other dependencies during the exploration process. Furthermore, the developer usually has to perform some tricks to control the explosion of relevant program elements although different types of program dependencies. Otherwise, he may easily get lost during exploration process.
Take the program snippet shown in Figure 8 as an example. A developer finds that the method ‘registerUser’ is relevant to the feature ‘register user’. He first navigates to the method ‘checkValidation’ called by registerUser and then further explores the program elements used by checkValidation. After that, he returns to the method registerUser and investigates other dependencies of registerUser, such as the methods ‘checkUserExistance’ and ‘addUser’. In this example, the developer considers registerUser as a focus point for the exploration-based extension.

It should be noted that developers do not always strictly follow single search or extension patterns in their feature location processes. They often use variants of the identified patterns or combine different patterns together (i.e., use hybrid patterns). Discussion about variant patterns and hybrid patterns can found in Section 7.

4.4. The conceptual framework for understanding feature location process

The empirical results of the first experiment suggest a hierarchical conceptual framework for understanding the feature location process. As shown in Figure 9, this conceptual framework consists of a collection of phases, patterns, and actions. A phase consists of collections of concrete actions directed toward the purpose of the phase. An action can be either physical or mental. In our study, we identified 11 types of physical actions and inferred six types of mental actions (see Section 4.1) that are important to understand the feature location process. We identified four phases with distinct purposes in the feature location process, that is, Seed Search, Extend, Validate, and Document. Furthermore, our study revealed that the actions are not independent of each other in a phase; they form various patterns in service of the purpose of the phases. We reported three such patterns in the Seed Search phase and two patterns in the Extend phase.

This conceptual framework provides an organized way to describe and understand feature location process. It indicates that feature location process engages several levels of information, including the purposes of developers’ actions over time, the patterns and strategies that developers adopt for accomplishing the purposes, the type of information that developers seek, and the tactics they utilize. Existing research on feature location has been mainly focused on the type of information that developers seek; little attention has been paid to the purposes and strategies of developers during feature location tasks and the concrete tactics that they utilize. Our conceptual framework could inspire more comprehensive studies of rigorous feature location processes.

This conceptual framework also allows researchers to start focusing on the human aspects of the feature location process. In the next section, we report our initial exploration in this direction. More specially, we taught a group of junior developers this conceptual framework and comparatively studied their performance during feature location tasks. This framework allowed us to better explain
the benefits and pitfalls of different feature location strategies and the trade-off of different tactics, so that developers could make more informed decisions in different situations. Although our results are by no means conclusive, they suggest that the proposed conceptual framework is promising in improving the developers’ effectiveness on feature location tasks.

5. THE IMPACT OF THE KNOWLEDGE OF FEATURE LOCATION PHASES, PATTERNS, AND ACTIONS ON FEATURE LOCATION PROCESS AND QUALITY

In this section, we evaluate the impact of the explicit feature location knowledge on the quality of feature location results using statistical analysis (Section 4.3.2). We also discuss our analysis of the impact of the feature location knowledge on feature location process (Sections 5 and 5.1.2).

5.1. The impact of explicit feature location knowledge on feature location quality

In the first experiment, our primary focus is to identify a variety of phases, patterns, and actions during feature location tasks. Thus, we advised the participants not to focus only on the quality of their feature location results. We encouraged them to use different approaches and strategies to complete the assigned tasks. Overall, the developers in the first experiment still achieved quite good results, 84.5% (±14.2%) precision, 78.9% (±11.8%) recall, and 80.6%

In the second experiment, we taught the knowledge of the identified feature location phases, patterns, and actions to two groups of junior developers. On the basis of their results in the two sessions, we investigated the impact of this knowledge on the performance of these junior developers on feature location tasks. In the third experiment, we recruited another two groups of junior developers. The objective of this experiment is to investigate the impact of self-learning on the performance of these junior developers for the same feature location tasks on the same subject systems.

We performed both longitudinal and lateral statistical comparisons to evaluate the impact of explicit feature location knowledge on the quality of feature location results. In the longitudinal analysis, we tested whether the participants’ performance had been significantly improved after learning the feature location knowledge by comparing the performance of the experimental groups ($T_1$ and $T_2$) in their before and after sessions. In the lateral analysis, we tested whether the improvement that resulted from feature location knowledge was significantly greater than the improvement that resulted from self-learning effect by comparing the improvement achieved by the experimental groups ($T_1$ and $T_2$) and the control groups ($T_3$ and $T_4$).

5.1.1. Hypotheses. Our longitudinal analysis compares the participants’ performance before and after they learned feature location knowledge. Note that we cannot compare the performance of the same developer on the same subject system and tasks before and after the developer learns feature location knowledge, because the developer would already know the subject system and tasks. Thus, our second experiment involved two equivalent groups of junior developers, $T_1$ and $T_2$. We compare the performance of these two equivalent groups of developers on the same subject systems and tasks, one without feature location knowledge and the other with feature location knowledge. In particular, we compare the performance of $T_1$s before session and $T_2$s after session on Buddi (denoted by $Before_{Buddi}(T_1)$ vs. $After_{Buddi}(T_2)$) and $T_2$s before session and $T_1$s after session on JGnash (denoted by $Before_{JGnash}(T_2)$ vs. $After_{JGnash}(T_1)$), respectively. The symmetrical design of our experiment ensures that the impact of the differences of learning capabilities of two groups of developers on the statistical analysis should be minimal.

We introduce three hypotheses for our longitudinal analysis as follows:

H$_{11}$: There is no difference between the recall of the participants before and after learning the feature location knowledge.
H$_{12}$: There is no difference between the precision of the participants before and after learning the feature location knowledge.
H$_{13}$: There is no difference between the $F$-measure of the participants before and after learning the feature location knowledge.
The alternative hypotheses of \( H_{11} \) to \( H_{13} \) ensure that there are statistically significant differences between the performance of the participants before and after learning the feature location knowledge.

Our lateral analysis compares the improvement of the participants’ performance that resulted from feature location knowledge and self-learning effect. For the two experimental groups (\( T_1 \) and \( T_2 \)) in the second experiment, we computed the performance difference on Buddi in the before and after sessions (Before\(_{\text{Buddi}}\)(\( T_1 \)) vs. After\(_{\text{Buddi}}\)(\( T_2 \))) and the performance difference on JGnash in the before and after sessions (Before\(_{\text{JGnash}}\)(\( T_2 \)) vs After\(_{\text{JGnash}}\)(\( T_1 \))). For the two control groups (\( T_3 \) and \( T_4 \)) in the third experiment, we computed the performance difference on Buddi in the before and after sessions (Before\(_{\text{Buddi}}\)(\( T_3 \)) vs. After\(_{\text{Buddi}}\)(\( T_4 \))) and the performance difference on JGnash in the before and after sessions (Before\(_{\text{JGnash}}\)(\( T_4 \)) vs. After\(_{\text{JGnash}}\)(\( T_3 \))). We then compared the performance difference on Buddi in the second experiment with that of Buddi in the third experiment (denoted by \( \text{Diff}_{\text{Buddi}}(\( T_1, T_2 \)) \) vs. \( \text{Diff}_{\text{Buddi}}(\( T_3, T_4 \)) \)) and the performance difference on JGnash in the second experiment with that of JGnash in the third experiment (denoted by \( \text{Diff}_{\text{JGnash}}(\( T_2, T_1 \)) \) vs. \( \text{Diff}_{\text{JGnash}}(\( T_4, T_3 \)) \)), respectively. Again, the symmetrical design of our experiments ensures that the impact of the differences of developers on the statistical analysis should be minimal.

We introduce three hypotheses for our lateral analysis as follows:

- **H21**: There is no difference in the improvement of recall between the groups learning the feature location knowledge (\( T_1 \) and \( T_2 \)) and the self-learning groups (\( T_3 \) and \( T_4 \)).
- **H22**: There is no difference in the improvement of precision between the groups learning the feature location knowledge (\( T_1 \) and \( T_2 \)) and the self-learning groups (\( T_3 \) and \( T_4 \)).
- **H23**: There is no difference in the improvement of F-measure between the groups learning the feature location knowledge (\( T_1 \) and \( T_2 \)) and the self-learning groups (\( T_3 \) and \( T_4 \)).

The alternative hypotheses of \( H_{21} \) to \( H_{23} \) ensure that there are statistically significant differences between the improvement of feature location performance that resulted from learning the feature location knowledge and the improvement that resulted from self-learning effect.

For all the hypotheses, we test the hypotheses on both Buddi and JGnash. We evaluate the hypotheses at a 0.05 level of significance.

### 5.1.2. Results of individual participants.

In the second and third experiments, the participants provided 1 to 17 methods for each task (7 on average), and none of them provided an empty result list for a task. Tables VI–IX list the precision, recall, and F-measure of all the participants in these two experiments. In the last line of each table, we provide the Shapiro–Wilk significance for each group of data (precision, recall, or F-measure). The Shapiro–Wilk test shows that all these data groups satisfy normal distribution.

The ‘before session’ and ‘after session’ columns of Tables VI and VII represent the two sessions of the second experiment (i.e., before and after the participants were taught feature location phases, respectively).
patterns, and actions), respectively. The before session and after session columns of Tables VIII and IX represent the two sessions of the third experiment.

Let us first investigate the quality of each participant’s results in the second experiment (Tables VI and VII) individually. Sixteen out of 18 participants had better $F$-measures in their after sessions compared with their before sessions. Among these 16 participants, eight had both better recalls and precisions, whereas

<table>
<thead>
<tr>
<th>Participant</th>
<th>Before session (JGnash)</th>
<th>After session (Buddi)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>Precision</td>
</tr>
<tr>
<td>P1</td>
<td>76.2%</td>
<td>87.3%</td>
</tr>
<tr>
<td>P2</td>
<td>66.7%</td>
<td>77.8%</td>
</tr>
<tr>
<td>P3</td>
<td>54.4%</td>
<td>81.4%</td>
</tr>
<tr>
<td>P4</td>
<td>68.0%</td>
<td>53.0%</td>
</tr>
<tr>
<td>P5</td>
<td>74.0%</td>
<td>34.0%</td>
</tr>
<tr>
<td>P6</td>
<td>58.9%</td>
<td>52.3%</td>
</tr>
<tr>
<td>P7</td>
<td>61.3%</td>
<td>77.2%</td>
</tr>
<tr>
<td>P8</td>
<td>32.2%</td>
<td>70.0%</td>
</tr>
<tr>
<td>P9</td>
<td>53.3%</td>
<td>62.1%</td>
</tr>
<tr>
<td>Average</td>
<td>60.5%</td>
<td>66.1%</td>
</tr>
<tr>
<td>Shapiro–Wilk significance</td>
<td>0.372</td>
<td>0.578</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Participant</th>
<th>Before session (Buddi)</th>
<th>After session (JGnash)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>Precision</td>
</tr>
<tr>
<td>P1</td>
<td>61.3%</td>
<td>83.2%</td>
</tr>
<tr>
<td>P2</td>
<td>73.4%</td>
<td>49.4%</td>
</tr>
<tr>
<td>P3</td>
<td>66.7%</td>
<td>67.8%</td>
</tr>
<tr>
<td>P4</td>
<td>36.1%</td>
<td>40.0%</td>
</tr>
<tr>
<td>P5</td>
<td>16.1%</td>
<td>45.6%</td>
</tr>
<tr>
<td>P6</td>
<td>54.4%</td>
<td>100.0%</td>
</tr>
<tr>
<td>P7</td>
<td>42.2%</td>
<td>65.3%</td>
</tr>
<tr>
<td>P8</td>
<td>32.2%</td>
<td>51.1%</td>
</tr>
<tr>
<td>P9</td>
<td>22.0%</td>
<td>55.6%</td>
</tr>
<tr>
<td>Average</td>
<td>44.9%</td>
<td>62.0%</td>
</tr>
<tr>
<td>Shapiro–Wilk significance</td>
<td>0.764</td>
<td>0.362</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Participant</th>
<th>Before session (JGnash)</th>
<th>After session (Buddi)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>Precision</td>
</tr>
<tr>
<td>P1</td>
<td>66.7%</td>
<td>77.8%</td>
</tr>
<tr>
<td>P2</td>
<td>76.2%</td>
<td>55.3%</td>
</tr>
<tr>
<td>P3</td>
<td>65.0%</td>
<td>87.0%</td>
</tr>
<tr>
<td>P4</td>
<td>33.3%</td>
<td>37.6%</td>
</tr>
<tr>
<td>P5</td>
<td>38.5%</td>
<td>28.6%</td>
</tr>
<tr>
<td>P6</td>
<td>58.9%</td>
<td>52.3%</td>
</tr>
<tr>
<td>P7</td>
<td>48.8%</td>
<td>48.3%</td>
</tr>
<tr>
<td>P8</td>
<td>49.8%</td>
<td>55.3%</td>
</tr>
<tr>
<td>P9</td>
<td>28.0%</td>
<td>65.0%</td>
</tr>
<tr>
<td>Average</td>
<td>51.7%</td>
<td>56.4%</td>
</tr>
<tr>
<td>Shapiro–Wilk significance</td>
<td>0.836</td>
<td>0.904</td>
</tr>
</tbody>
</table>
Table X. Results of $t$-tests of the hypotheses.

<table>
<thead>
<tr>
<th>H</th>
<th>Approach</th>
<th>Samples</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>$\mu$</th>
<th>$\sigma^2$</th>
<th>$DF$</th>
<th>$PC$</th>
<th>$T$</th>
<th>$T_{crit}$</th>
<th>$p$</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{11}$</td>
<td>Before_Buddi($T_1$)</td>
<td>9</td>
<td>0.367</td>
<td>0.778</td>
<td>0.544</td>
<td>0.563</td>
<td>0.018</td>
<td>8</td>
<td>0.920</td>
<td>-8.356</td>
<td>2.306</td>
<td>3.19E-05</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>After_Buddi($T_2$)</td>
<td>9</td>
<td>0.533</td>
<td>0.880</td>
<td>0.721</td>
<td>0.725</td>
<td>0.010</td>
<td>8</td>
<td>0.926</td>
<td>-3.296</td>
<td>2.306</td>
<td>0.011</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Before_JGnash($T_1$)</td>
<td>9</td>
<td>0.322</td>
<td>0.762</td>
<td>0.613</td>
<td>0.606</td>
<td>0.018</td>
<td>8</td>
<td>0.926</td>
<td>3.963</td>
<td>2.306</td>
<td>0.004</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>After_JGnash($T_2$)</td>
<td>9</td>
<td>0.524</td>
<td>0.830</td>
<td>0.683</td>
<td>0.668</td>
<td>0.009</td>
<td>8</td>
<td>0.944</td>
<td>-2.453</td>
<td>2.306</td>
<td>0.040</td>
<td>Reject</td>
</tr>
<tr>
<td>$H_{12}$</td>
<td>Before_Buddi($T_1$)</td>
<td>9</td>
<td>0.433</td>
<td>1.000</td>
<td>0.800</td>
<td>0.749</td>
<td>0.037</td>
<td>8</td>
<td>0.926</td>
<td>4.342</td>
<td>2.306</td>
<td>0.002</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>After_Buddi($T_2$)</td>
<td>9</td>
<td>0.590</td>
<td>0.870</td>
<td>0.730</td>
<td>0.729</td>
<td>0.010</td>
<td>8</td>
<td>0.959</td>
<td>4.087</td>
<td>2.306</td>
<td>0.004</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Before_JGnash($T_1$)</td>
<td>9</td>
<td>0.340</td>
<td>0.873</td>
<td>0.700</td>
<td>0.661</td>
<td>0.030</td>
<td>8</td>
<td>0.944</td>
<td>-4.087</td>
<td>2.306</td>
<td>0.004</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>After_JGnash($T_2$)</td>
<td>9</td>
<td>0.570</td>
<td>0.760</td>
<td>0.703</td>
<td>0.691</td>
<td>0.003</td>
<td>8</td>
<td>0.930</td>
<td>-4.342</td>
<td>2.306</td>
<td>0.002</td>
<td>Reject</td>
</tr>
<tr>
<td>$H_{13}$</td>
<td>Before_Buddi($T_1$)</td>
<td>9</td>
<td>0.380</td>
<td>0.780</td>
<td>0.584</td>
<td>0.584</td>
<td>0.016</td>
<td>8</td>
<td>0.959</td>
<td>4.342</td>
<td>2.306</td>
<td>0.002</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>After_Buddi($T_2$)</td>
<td>9</td>
<td>0.570</td>
<td>0.760</td>
<td>0.703</td>
<td>0.691</td>
<td>0.003</td>
<td>8</td>
<td>0.959</td>
<td>4.087</td>
<td>2.306</td>
<td>0.004</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Before_JGnash($T_1$)</td>
<td>9</td>
<td>0.360</td>
<td>0.780</td>
<td>0.599</td>
<td>0.602</td>
<td>0.013</td>
<td>8</td>
<td>0.930</td>
<td>-4.087</td>
<td>2.306</td>
<td>0.004</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>After_JGnash($T_2$)</td>
<td>9</td>
<td>0.550</td>
<td>0.770</td>
<td>0.694</td>
<td>0.675</td>
<td>0.005</td>
<td>8</td>
<td>0.930</td>
<td>-4.342</td>
<td>2.306</td>
<td>0.002</td>
<td>Reject</td>
</tr>
<tr>
<td>$H_{21}$</td>
<td>Diff_Buddi($T_1, T_2$)</td>
<td>9</td>
<td>-0.057</td>
<td>0.300</td>
<td>0.203</td>
<td>0.162</td>
<td>0.012</td>
<td>8</td>
<td>0.898</td>
<td>2.589</td>
<td>2.306</td>
<td>0.032</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Diff_Buddi($T_3, T_4$)</td>
<td>9</td>
<td>-0.096</td>
<td>0.298</td>
<td>0.123</td>
<td>0.116</td>
<td>0.015</td>
<td>8</td>
<td>0.917</td>
<td>2.313</td>
<td>2.306</td>
<td>0.049</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Diff_JGnash($T_1, T_2$)</td>
<td>9</td>
<td>-0.162</td>
<td>0.256</td>
<td>0.065</td>
<td>0.062</td>
<td>0.015</td>
<td>8</td>
<td>0.917</td>
<td>2.313</td>
<td>2.306</td>
<td>0.049</td>
<td>Reject</td>
</tr>
<tr>
<td>$H_{22}$</td>
<td>Diff_Buddi($T_1, T_2$)</td>
<td>9</td>
<td>-0.191</td>
<td>0.144</td>
<td>-0.105</td>
<td>-0.050</td>
<td>0.012</td>
<td>8</td>
<td>0.495</td>
<td>-0.194</td>
<td>2.306</td>
<td>0.851</td>
<td>Accept</td>
</tr>
<tr>
<td></td>
<td>Diff_Buddi($T_3, T_4$)</td>
<td>9</td>
<td>-0.420</td>
<td>0.333</td>
<td>-0.160</td>
<td>-0.126</td>
<td>0.045</td>
<td>8</td>
<td>0.495</td>
<td>-0.194</td>
<td>2.306</td>
<td>0.851</td>
<td>Accept</td>
</tr>
<tr>
<td></td>
<td>Diff_JGnash($T_1, T_2$)</td>
<td>9</td>
<td>-0.467</td>
<td>0.146</td>
<td>-0.143</td>
<td>-0.113</td>
<td>0.032</td>
<td>8</td>
<td>0.636</td>
<td>1.756</td>
<td>2.306</td>
<td>0.117</td>
<td>Accept</td>
</tr>
<tr>
<td>$H_{23}$</td>
<td>Diff_Buddi($T_1, T_2$)</td>
<td>9</td>
<td>-0.194</td>
<td>0.390</td>
<td>0.042</td>
<td>0.068</td>
<td>0.035</td>
<td>8</td>
<td>0.636</td>
<td>1.756</td>
<td>2.306</td>
<td>0.117</td>
<td>Accept</td>
</tr>
<tr>
<td></td>
<td>Diff_Buddi($T_3, T_4$)</td>
<td>9</td>
<td>-0.318</td>
<td>0.214</td>
<td>-0.020</td>
<td>-0.022</td>
<td>0.030</td>
<td>8</td>
<td>0.961</td>
<td>2.443</td>
<td>2.306</td>
<td>0.040</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Diff_JGnash($T_1, T_2$)</td>
<td>9</td>
<td>-0.079</td>
<td>0.275</td>
<td>0.122</td>
<td>0.106</td>
<td>0.011</td>
<td>8</td>
<td>0.961</td>
<td>2.651</td>
<td>2.306</td>
<td>0.029</td>
<td>Reject</td>
</tr>
<tr>
<td></td>
<td>Diff_JGnash($T_3, T_4$)</td>
<td>9</td>
<td>-0.144</td>
<td>0.233</td>
<td>0.074</td>
<td>0.072</td>
<td>0.011</td>
<td>8</td>
<td>0.961</td>
<td>2.651</td>
<td>2.306</td>
<td>0.029</td>
<td>Reject</td>
</tr>
</tbody>
</table>

The measurements are reported in the columns are the following: minimum value, maximum value, median, means ($\mu$), variance ($\sigma^2$), the degrees of freedom ($DF$), the Pearson correlation coefficient ($PC$), $T$ statistics, $T_{crit}$ and the statistical significance ($p$).
the other eight had better recall but worse precision or better precision but worse recall. In the post-experiment questionnaire, the participants indicated that the knowledge of feature location phases, patterns, and actions helped their completion of the assigned tasks. We further investigated the performance of participants with different program experience. On the basis of the participants’ pre-experiment survey, we identified 8 experienced developers and 10 less experienced in the second experiment. Although the performance of experienced developers is always better than that of less experienced, the gap between them was reduced, after they learned the knowledge of feature location phases, patterns, and actions.

Now let us investigate the quality of each participant’s results in the third experiment (Tables VIII and IX) individually. Twelve out of 18 participants had better $F$-measures. Among these 12 participants, five had both better recalls and precisions, whereas the other seven had better recall but worse precision or better precision but worse recall. It can be seen that the performance of most participants had been improved to a certain extent in the after session, as they may accumulate certain experience in the completion of the feature location tasks in the before session.

5.1.3. Results of hypotheses testing. We use paired sample $t$-tests to evaluate the null hypothesis $H_{11}$ to $H_{13}$ on both Buddi and JGnash. The results of these six tests are shown in Table 10. On the basis of the results, we reject $H_{11}$ to $H_{13}$ and accept their alternative hypothesis, that is, there are statistically significant differences between the performance of the participants before and after learning the feature location knowledge. The test results further show that all the measures increased after learning the feature location knowledge except the decrease of the participants’ precision on Buddi. Our inspection suggests that this decrease was because after learning the feature location knowledge, the participants were able to use various strategies for finding relevant program elements and thus returned more results in the after session. This factor is reflected by the significant increase of recall and $F$-measure that the participants achieved on Buddi. In terms of the significant improvement on $F$-measure, our data show that the participants achieved better feature location results after learning the feature location knowledge.

Similarly, we use paired sample $t$-tests to evaluate the null hypothesis $H_{21}$ to $H_{23}$ on both Buddi and JGnash. The results of these six tests are shown in Table 10. On the basis of the results, we reject $H_{21}$ and $H_{23}$ and accept their alternative hypothesis and at the same time accept $H_{22}$. This indicates that there are statistically significant differences between the improvement of feature location performance that resulted from learning the feature location knowledge and the improvement that resulted from self-learning effect in terms of recall and $F$-measure, but not in terms of precision. The test results further show that the participants can achieve greater improvement on recall and $F$-measure by learning feature location knowledge. In terms of the greater improvement on $F$-measure, our data show that the improvement of the participants’ performance that resulted from the feature location knowledge is greater than that resulted from self-learning effect.

5.2. The choices of search and extension patterns

Let us now examine how often each search pattern (identified in Section 4.3.1) and extension pattern (identified in Section 4.3.1.1) has been adopted during feature location tasks and what their potential impact is on the quality of feature location results.

5.2.1. The choices of search pattern. Tables XI and XII summarize the statistics we collected for the choices of different search patterns and different extension patterns for the second experiment. Note that due to the objective and design of the first experiment, we deem that the statistics of the pattern usage in that experiment would be biased. Thus, we did not include them in the discussion of this paper.

As expected, IR-based pattern has been adopted most frequently (by 10 participants) in the second experiment, whereas the adoption of exploration-based and execution-based patterns was about the same. This indicates that participants in our experiment tended to start their feature location tasks by searching for the entrance points based on the perceived relevant keywords. Note that in the before session of the second experiment, participants used these three patterns ‘subconsciously’ without explicit knowledge about them.
We expected that participants would use different search patterns for different categories of tasks and/or different subject systems. However, we did not observe such phenomena in our experiment. Furthermore, we did not observe the dramatic changes in the adoption of different patterns before and after the participants were taught these patterns. Participants seem to have a consistent preference for the searching strategies that they would like to use.

Only in the after session of the second experiment, two participants adopted different patterns from the one they regularly adopted. Our post-experiment interview confirmed that one of these two participants was inspired by the variety of search patterns and would like to try something different. The F-measure of this participant in the before session of the second experiment ranked second among the 18 participants. For the second participant who adopted a different pattern, he adopted execution-based pattern first but failed to find good entrance points. And then, he changed to use IR-based pattern and successfully found the relevant entrance point.

Although participants did not change the search patterns that they prefer to use, the overall quality (F-measure) of their feature location results was improved (Table XI), after they explicitly learned the knowledge of feature location phases, patterns, and actions. We attributed this improvement to the more detailed understanding of what steps are involved in the feature location process and how they could proceed in different situations. Section 5.1.2 has a further discussion on this.

Finally, we observed the differences in the result quality of participants who adopted different patterns. However, our analysis suggested that we cannot simply attribute this difference to the effectiveness of different patterns, because we found out that experienced developers in our experiment tended to use execution-based or exploration-based patterns. These developers usually performed better than others. We cannot determine whether the developers’ programming experience or the adopted pattern is the primary factor that affects the quality of their feature location results.

5.2.2. The choices of extension pattern. Exploration-based extension pattern has been adopted most frequently (by 10 participants), whereas six participants adopted execution-based extension pattern. Two participants did not perform any extension action in the before session of the second experiment, and both of them adopted exploration-based extension pattern in the after session.

Similar to the adoption of different search patterns, we did not observe the dramatic changes in the adoption of different extension patterns before and after the participants were taught these patterns. Participants seem to have a consistent preference for the extension strategies that they would like to use as well. Actually, we found only two participants who did no extension in their before sessions.
changed to use exploration-based extension in the three tasks of their after sessions, making six more uses of exploration-based extension pattern (Table XII).

We expected that participants who adopt execution-based or exploration-based search pattern would be inclined to use execution-based or exploration-based extension pattern, respectively. However, we did not observe such phenomena in our experiment. It seems that there is no direct relationship between the adoption of execution-based or exploration-based search strategies and extension strategies.

Similar to the choices of search patterns, although participants did not change the extension patterns that they prefer to use, the overall quality (recall and $F$-measure) of their feature location results was improved (Table XII).

5.3. How developers perform feature location, before versus after?

Our analysis of the feature location processes of 36 participants in the second and third experiments suggests that the explicit knowledge of the feature location phases, search/extension patterns, and physical/mental actions improves the participants’ ability to use these phases, patterns, and actions in a more explicit, complete, and systematic way. This ability plays a significant role in improving the efficiency of feature location process and the quality of feature location results.

The feature location knowledge affected the feature location process and quality mainly from two perspectives. First, the feature location processes of the participants in the after session of the second experiment more likely involved four phases distinctively (from Seed Search → Extend → Validate → Document). The participants seemed to be more aware of what goals they should achieve in different phases. Second, these participants seemed to use search and extension patterns in a more complete and systematic way. Furthermore, they more likely chose appropriate patterns and actions that better fit the task properties, and they paid more attention to the in-process feedbacks during feature location process. However, we did not observer the similar phenomena in the feature location processes of the participants in the third experiment.

Let us illustrate our findings with the feature location processes of one of the junior participants from our second experiment. In the before session of the second experiment, this developer was asked to work on three feature location tasks on the subject system Buddi. One of the three tasks is ‘Add New Account’. After the tutorial session, he was asked to work on another three feature location tasks on JGnash in the after session. One of these three tasks, that is, Add New Transaction, is similar to the task Add New Account in the before session: The nature and size of the subject systems are similar (i.e., personal finance application); both subject systems have relevant UI elements (e.g., the menu item ‘Create Account’ or ‘Recurring Transaction’) that contain useful hints for the tasks; the nature of both tasks are similar (database-related), and they have similar complexity (e.g., the number of relevant program elements in the subject systems).

In the before session, the feature location process of this participant was ad hoc. He did not seem to have a clear understanding of how he should start and how he should proceed during feature location. His feature location process did not manifest distinctive phases. For example, we cannot clearly distinguish whether he attempted to search for entrance points or to extend the initial search results. Furthermore, the use of searching and extension patterns was often incomplete. For example, he adopted exploration-based extension pattern to identify more relevant program elements. However, he was quickly lost during the exploration process, because he followed call relations too far away from the starting focus point and failed to get back to the focus point to explore other relations from it. Finally, his use of feature location actions seemed random. For example, he adopted IR-based search pattern to identify entrance points. He performed four file searches. But the keywords that he used for the search seemed very random, from ‘create account’ (0 match) to ‘account’ (1225 matches), ‘account name’ (1 match), and ‘addaccount’ (24 matches). He did not utilize hints from previous search results very well.

In contrast, in the after session, the feature location process of this developer manifested much more distinctive phases. For example, he first used IR-based technique to identify 10 method declarations. And then, he attempted to extend from these 10 method declarations one by one in order to identify more relevant program elements. It seems that he knew clearly what he would like to achieve before taking actions. Furthermore, his use of search and extension patterns was much more complete and systematic. Take the extension phase as example again. He still adopted exploration-based extension
pattern. But this time, he obviously learned that he needed to extend from several different focus points and needs to get back to the starting focus point and consider different types of static dependencies. As a result, he nearly identified all the relevant program elements (92.9% recall) for the task Add New Transaction. Finally, he seemed to know how to make good use of the in-process feedbacks for his subsequent actions. For example, he adopted IR-based search pattern again for identifying seed entrance points. But this time, he first performed a file search using the keyword ‘transaction’. And then, he quickly went through the returned program elements. In his second search, he refined his keyword as a regular expression ‘add*transaction’ and used Java search for method declaration. This Java search returned 10 method declarations that he used as entrance points for further extension. Obviously, he successfully picked the hints from the results of the first file search.

6. THE IMPACT OF EXTERNAL FACTORS ON FEATURE LOCATION PROCESS

In addition to the quantitative analysis of the impact of the inherent feature location phases, patterns, and actions on feature location process and quality, our observations on the screen-recorded videos of participants’ feature location processes, their post-experiment surveys, and the interviews with participants identify three key external factors that may affect their choices and usage of different feature location patterns and actions during feature location tasks. These three external factors are as follows:

- ‘Human factors’ refer to factors related to the capability, programming experience, and the personal preference of developers. It also includes the developers’ familiarity with the programming language and the development environment.
- ‘Task properties’ refer to factors related to the inherent properties of a given feature location task, including the nature of the subject system (e.g., web-based information system vs. desktop application), the characteristics of the features under investigation (e.g., whether the feature is UI-related or algorithm-related and whether the feature description suggests relevant keywords), and the purposes of the feature location task (e.g., bug fixing vs. feature enhancement).
- ‘In-process feedbacks’, in contrast to task properties, refer to factors related to different kinds of feedbacks obtained during the feature location process, for example, the number of times of IR-based searches with different keywords, the number of program elements returned by IR-based search, the depth of the exploration-based extension from the initial focus point, and whether the breakpoint has been reached.

6.1. The impacts of external factors on feature location patterns

Let us first examine how the three external factors affect the developers’ choices of search (see Section 4.3.1) and extension patterns (see Section 4.3.1.1) and how these factors affect the way that developers perform the chosen patterns.

Human factors have the most significant impact on the developers’ choices on search and extension patterns. As discussed in Section 5, a developer’s capability, programming experience, and personal preference usually determine his choices of search and extension patterns. Furthermore, experienced developers who are more familiar with the program-language constructs and the development environment tend to use more patterns than less experienced developers. For example, in our experiment, experienced developers often start with exploration-based search pattern to obtain some initial understanding of the subject systems and to narrow down to some places (e.g., packages or classes that are potentially relevant) where they may then use IR-based or execution-based search patterns. In contrast, less experienced developers usually stick to the search pattern they prefer. In our experiment, they often chose IR-based search pattern and repetitively used the same pattern even if the pattern brings no success. Human factors also affect the ways that a developer performs the chosen pattern. For example, when using exploration-based search/extension patterns, experienced developers usually were able to explore much deeper without getting lost than less experienced developers, because experience developers can more effectively use the features of development environment, such as adding to-dos to keep track of their exploration.
In addition to human factors, task properties can also affect the developers’ choices on search patterns. For example, if the description of a given feature location task or the UI of the subject system provides useful hints for a developer to conceive suitable keywords, he is highly likely to use IR-based search pattern. On the other hand, if the given feature can be triggered from the UI of the subject system, the developer may consider using execution-based search pattern. Or, if a feature involves clearly some specific program elements (e.g., packages, classes, or files), the developer is likely to use exploration-based search pattern after he explores the package structure.

Compared with human factors and task properties, in-process feedbacks have the least impact on the choices and usage of search and extension patterns. This is because in-process feedbacks mainly affect how developers take subsequent actions based on the results of his earlier actions. For example, when several file searches keep returning too many program elements, the developer may consider using a more advanced Java search. Thus, the impact of in-process feedbacks is at a more fine-grained level than patterns, that is, actions. Section 5.2 has a further note on this.

Note that the impact of the human factors and task properties on the choices and usage of search patterns is larger than their impacts on extension patterns, because search patterns are more sensitive to the developer’s experience and preference as well as the inherent properties of the task, whereas extension patterns mainly depend on whether static or dynamic program analysis techniques are used for extension.

6.2. The impacts of external factors on feature location actions

Let us now examine how the three external factors affect the ways that developers perform specific actions. As discussed in Section 4.1, the elementary actions that we identified abstract the concrete actions that developers perform during feature location process. The same elementary action may have several variants. For example, for the action ‘identify relevant keywords’, a developer may identify the keywords from feature description or UI. Furthermore, he may use plain texts or regular expressions to represent the keywords.

Human factors still have more significant impact on the ways that developers perform actions than task properties and in-process feedbacks. Generally speaking, experienced developers tend to use more advanced mechanisms when performing specific actions. For example, they often set more restrictive conditions (e.g., search by declaration or by reference) and/or use regular expression when performing the ‘search program elements’ action; they tend to set appropriate conditional breakpoints to speed up the ‘step program’ action. Furthermore, experienced developers tend to switch alternative ways of performing a specific action more frequently. For example, when performing the search program elements action, experienced developers often switch between ‘Java Search’ and ‘File Search’, whereas less experienced developers usually stick to File Search only.

Task properties have little impact on the ways that developers perform specific actions, whereas the in-process feedbacks have some impacts on the actual actions that developers perform, because developers often adjust their ways of performing subsequent actions based on the results obtained from earlier actions. For example, when performing the action enrich/refine keywords, developers may change to use different words or synonyms if they obtain too few results in the last search. On the other hand, if the last search returns too many results, they will attempt to enrich keywords, change to use more specialized keywords, and/or more restrictive search options. Furthermore, our experiments indicate that experienced developers can better adjust their choices of actions and how they perform specific actions based on in-process feedbacks.

7. DISCUSSION

Conducting this exploratory study has given us some interesting insights into the process of feature location, the techniques for feature location, and relevant tool support. We discuss them in this section.

First, helping developers explore and understand code has been an important challenge in software engineering research. Several tools have been built to support the developers’ work in different phases of the proposed conceptual framework, such as Robillard and Murphy’s work on concern graph [25]
HOW DEVELOPERS PERFORM FEATURE LOCATION TASKS

and relevant tools [26–28]. Our conceptual framework for feature location processes could provide a foundation for a better integration of existing tools.

Second, as discussed in Sections 4.3.1 and 4.3.1.1, different search and extension patterns primarily explore different sources of information and utilize different tactics, such as textual keywords and program search in IR-based pattern, execution scenarios and program debugging in execution-based patterns, and static dependencies and program navigation in exploration-based patterns. However, these patterns also seek other supportive sources of information and tactics. This finding gives rise to the need for a better integration of different sources of information and different searching, debugging, and exploring tactics during feature location tasks.

Furthermore, our observations showed that developers did not always follow exactly the search or extension patterns that they chose, even after they had learnt those search and extension patterns. They often used variants of the identified patterns or combined different patterns together in their feature location processes. In fact, some patterns involve variation points such as optional actions in IR-based search pattern (Figure 3). We also found that developers may combine different patterns together, that is, use hybrid patterns. In this case, one dominant pattern was usually used as the main strategy, and some other patterns were used for assistance. For example, developers who used execution-based or exploration-based search patterns often used IR-based search pattern to quickly narrow the scope of possible files to be further explored. This kind of combination usually can help developers reach the desired program elements more quickly and accurately. Another observation was that experienced developers were more likely to use hybrid patterns than less experienced developers.

Third, as discussed in Section 4.2, because of the limitations of the development environment and tools, developers usually simply record the names of program elements relevant to the given feature in the Extend and Document phases. The lack of documentation mechanisms increases the difficulty in keeping track of the relevant program elements during feature location process, inferring the purposes, rationales, and strategies of the developers’ actions, and sharing and reusing previous feature location results and/or experiences of the task at hand. This gives rise to the need for a process-oriented documentation mechanism that can record more detailed information about feature location process, rationales, and results. Note that this documentation mechanism is different from artifact-oriented task organization mechanism, for example, Mylyn [29], which focuses on capturing, modeling, and persisting elements and relations relevant to a task.

For example, in addition to recording individually program elements relevant to a given feature, the interdependencies between these program elements [26] may also be useful for understanding and modifying the feature. The detailed information about the feature location actions, such as different keywords used for search, different kinds of static dependencies explored from a focus point, and the positions where breakpoints are set, would also be very important for the completion of the feature location task. How developers start and proceed during feature location, such as what patterns they use, when they switch patterns, and the ordering and frequencies of their actions, is another piece of important information for understanding feature location process, especially for inferring the purposes, rationales and strategies behind the developers’ actions during feature location. If all such additional information about feature location process and results could be properly utilized, for example, in the smart feature location wizard discussed later, we would be able to provide better support for developers in feature location or other software maintenance tasks.

Fourth, our conceptual framework suggests some guidelines for developers to perform feature location tasks. First, developers need to be aware of distinct phases and their purposes during feature location process so that they can adopt appropriate searching strategies and actions that align well with the purpose of the phase. For example, finding an accurate entrance point is the main purpose of Seed Search phase, whereas achieving a good coverage is the most important purpose of Extend phase. Second, developers need to understand various applicable search and extension patterns and be able to flexibly select, tailor, or combine them on the basis of different task contexts. For example, IR-based search pattern may be used before execution-based or exploration-based search patterns to quickly narrow down the scope of possible files to be further executed or explored. Third, developers need to understand and improve their skills required for performing different actions, such as quickly filtering irrelevant classes by package and/or class name, and determining the relevance of returned results according to the location of keywords in the search results. These skills can significantly improve the efficiency of feature location.
Fifth, our detailed understanding of feature location processes could lead to even greater tool support. We envision a possibility to develop an intelligent, interactive, online feature location wizard that can offer more contextual-sensitive and personalized support for the developers’ feature location tasks. For example, this wizard could monitor and analyze the developer’s actions on the fly. Unlike existing tools [26] that mainly focus on the program information, our wizard would attempt to infer the purpose of developer’s actions, based on, for example, the temporal ordering and the frequency of the developer’s actions. Understanding the purpose behind these actions would allow it to provide more contextual-sensitive support for what developers are currently working on.

Furthermore, as discussed in Section 5, developers may tend to have their consistent preference for certain patterns/strategies during feature location tasks. After observing the developer’s activities for a certain period of time, the wizard may learn a probabilistic model of developer’s behavior. This model would integrate not only the investigation history of the developer but also other important information such as the pattern that the developer prefers and the analysis phase that he is currently in. On the basis of this model, the wizard could then offer personalized guided navigation support for the developer’s work. For example, it could pre-fetch and cache the program elements that the developer would highly likely investigate in the near future. By summarizing and presenting these elements in an intuitive way, the wizard could greatly improve the efficiency of the developer’s work, especially in Seed Search and Extend phases.

Finally, as shown in Sections 4.3.1 and 4.3.1.1, feature location phases consist of various patterns involving many interactive physical and/or mental actions. This intelligent wizard could interactively help the developer when he cannot make progress in his tasks. For example, the wizard may observe that the developer repeats a cycle of searching program elements and tries too many results refining keywords but seems not to have a clear target. In such cases, the wizard could prompt some keywords based on the syntactic and/or semantic analysis of code. Or it could suggest some alternative strategies, for example, using the other search patterns that may be potentially applicable.

8. THREATS TO VALIDITY

Our findings are subject to a number of limitations in the design of our study. We studied the developers’ work in controlled experiments instead in a real-world context. Although we recruited developers with industrial experience, adopted four real-world subject systems, and designed three categories of realistic tasks, the limited number of subject systems and tasks and the limited diversity of developers may limit the generalizability of our study. Furthermore, these studies were performed on relatively small systems with few professional developers, so all our findings may not be applicable on systems with millions of LOC that are maintained by teams of professional developers.

Participants in our study were asked to work independently and on unfamiliar systems. Developers in industry usually work in teams and are more or less familiar with the code they are responsible for. Further studies are required to generalize our findings in such collaborative context and with developers who have sufficient domain knowledge and familiarity with the subject systems.

The use of the think-aloud protocol in the feature location processes of some participants may cause Hawthorne effect [21], that is, the participants’ knowledge that they are in an experiment modifies their normal behavior. This effect may influence the results of our analysis of participants’ behaviors.

The length of experiment session (60 min) is somewhat arbitrary. It was based on our understanding of the complexity of the subject systems, the difficulty of the tasks that we designed, and a pre-experiment pilot study involving two additional developers (one experienced and one less experienced). In the post-experiment questionnaires, participants rated the difficulty of our feature location tasks at 3.42 (±1.53) (1 is very difficult, and 5 is very easy). On the basis of the screen videos of the participants’ work, we observed that about 75% of the participants completed their assigned tasks within 45–55 min. However, this 60-min time constraints may pressure the participants to complete all of the tasks as fast as possible and thus affect their actions during the experiments.

In the second experiment, we observed certain improvements in the quality of the participants’ feature location results. However, because of the short period of time that participants had to absorb the knowledge of feature location phases, patterns, and actions that we taught, their improvement is relatively
minor, especially in the bigger subject system JGnash. We speculate that their performance might manifest more significant improvements, had they been given a practice session before the real experiment.

Many findings in this study were based on our subjective interpretations of the full-screen videos of the developers’ work. The identification of feature location actions, patterns, and phases was based on our subjective interpretations of the full-screen videos of the developers’ work. Our experience in this study suggested that the most difficult part is to infer the intention behind the observable actions of developers and categorize them into elementary actions. The inference of intentions is especially difficult for those similar-looking but intentionally different actions. For example, the action that a developer navigates from one method to another does not always indicates that he is exploring static dependency. In contrast, the developer is trying to learn more about a method by reading its related methods. Thus, his navigation action should be categorized as ‘Read code’. This inference usually can be supported by an additional observation that the developer frequently returns back to the focus method in a short period of time.

The complexity of the developers’ actions during feature location tasks may incur errors in our analysis. This impacts the identification and analysis of feature location phases, patterns, actions, and their interpretations. Some advanced measurement and analysis tools, for example, Automated In-process Software Engineering Measurement and Analysis tools introduced in [30], may help to improve our analysis on feature location actions, patterns, and phases, because these tools can capture the developers’ behaviors and related artifacts in a nonintrusive manner. However, this kind of tools has its own limitation in that they can only capture the developers’ raw behaviors by discrete events and do not offer much help in inferring high-level actions from raw behavior.

Another source of subjectiveness is the identification of ground truth of traceability links that impacts the evaluation of the quality of participants’ feature location results. Furthermore, the intrinsic imprecision or subjectivity in our measurement of feature location performance in terms of precision, recall, and F-measure may also influence the validity of our findings.

In this study, we considered only one programming language (Java) and one development environment (Eclipse). Some of our findings, such as search patterns, would be different if other languages and environments were used. For example, different languages may support different types of dependencies; different development environments may summarize and present code in different ways.

A major threat to our statistical analysis on the comparison between the performance improvement that resulted from feature location knowledge and that resulted from self-learning effect lies in the fact that the experimental groups (T1 and T2) and the control groups (T3 and T4) may not be strictly equivalent in their capability (Table II). To address this threat, we had tried our best to make the control groups comparable with the experimental groups. From Table II, we can see that most participants in T3 and T4 except two have the same capability scores as their counterparts in T1 and T2. Furthermore, instead of comparing their performance directly, we compared the performance improvement that resulted from feature location knowledge in the experimental groups with the improvement that resulted from self-learning effect in the control groups. By using performance improvement, that is, the differences of the participants’ performance of feature location results in their before and after sessions, the impact of this threat on the statistical analysis can also be alleviated.

Finally, we only conducted a statistical analysis for the improvement on feature location performance after learning the knowledge of feature location phases, patterns, and actions. The other findings, for example, those about the choices and usage of patterns, and the impact of external factors, are based on our qualitative observation and analysis of screen-captured videos, post-experiment questionnaires, and interviews. This is because these findings largely depend on subjective interpretation about human behaviors and other human-oriented characteristics, which can often only be interpreted qualitatively in human studies. However, our observations and interviews involved a few hundred hours of human efforts. Although such findings are qualitative, we believe they are still important to understand and interpret the developers’ behavior in feature location process.

9. CONCLUSION AND FUTURE WORK

In this paper, we reported an exploratory study of feature location processes with three experiments, involving 56 developers, 4 real-world subject systems, and 12 feature location tasks. On the basis of the
empirical results of this study, we proposed a conceptual framework for understanding feature location processes, which consists of a collection of phases, patterns, and actions. This framework allows us to further investigate the human aspects of the feature location process. Our initial exploration suggested that this conceptual framework can be effectively imparted to junior developers and consequently improve their performance on feature location tasks. In addition to the impact of this conceptual framework on feature location process, our study also showed that three key external factors, that is, human factors, task properties, and in-process feedbacks, also affect the choices and usage of different feature location patterns and actions.

In the future, we plan to apply data mining techniques to automatically analyze the scripts of the developers’ actions transcribed from the screen-recorded videos of their feature location process. Such mining techniques may help us identify more recurring patterns in feature location process. Furthermore, we are very interested in developing an intelligent, interactive, online feature location wizard, based on our findings in this work.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and suggestions that allowed us to improve our paper. This work is supported by the National High Technology Development 863 Program of China under grant no. 2012AA011202 and the National Research Foundation for the Doctoral Program of Higher Education of China under grant no. 20100071110031.

REFERENCES

18. Rajlich V. Intensions are a key to program comprehension. Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC), 2009; 1–9.

**AUTHORS’ BIOGRAPHIES:**

**Jinshui Wang** is a PhD student of the School of Computer Science at Fudan University, China. He received his master’s degree from Xiamen University in 2009. His PhD work mainly concerns software maintenance and reverse engineering.

**Dr. Xin Peng** (http://www.se.fudan.edu.cn/pengxin/) is an associate professor of the School of Computer Science at Fudan University, China. He received his PhD degree in computer science from Fudan University in 2006. His current research interests include requirements engineering, self-adaptive system, software product line, software maintenance, and reverse engineering. His paper won the Best Paper Award at the Software Maintenance 2011 conference (ICSM 2011).
Dr. Zhenchang Xing is an assistant professor in the School of Computer Engineering, Nanyang Technological University (NTU). Before joining NTU, he was a Lee Kuan Yew Postdoctoral Fellow in the School of Computing at the National University of Singapore. He received his PhD in computer science from University of Alberta, Canada, in 2008, and his MEng and BS degrees from Nankai University, China, in 1997 and 2000. Dr. Xing’s research interests include human-centric software development infrastructure and tools, sustainable software maintenance and reengineering, and reliable and secure system operation. His recent study on the behavioral patterns of developers in feature location activities has received the Best Paper Award at the 27th International Conference on Software Maintenance (2011). His research paper on refactoring practices has received the Best Paper Award at the 22nd International Conference on Software Maintenance (2006). He regularly serves on the program committees of IEEE ICSM, WCRE, ICPC, and IBM CASCON.

Wenyun Zhao is a full professor of the School of Computer Science at Fudan University, China. He received his master’s degree from Fudan University in 1989. His current research interests include software reuse, software product line, software component, and architecture.