An Exploratory Study of Feature Location Process

Distinct Phases, Recurring Patterns, and Elementary Actions

Jinshui Wang[1], Xin Peng[1], Zhenchang Xing[2], Wenyun Zhao[1]
[1]School of Computer Science and Technology, Fudan University, Shanghai, China
[2]School of Computing, National University of Singapore, Singapore
{09110240018, pengxin, wyzhao}@fudan.edu.cn
xingzc@comp.nus.edu.sg

***Abstract*—Developers often have to locate the parts of the 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- and knowledge-intensive process. Researchers have investigated information retrieval, static and dynamic analysis based techniques to assist developers in such feature location activities. However, little work has been done on better understanding how developers perform feature location tasks. In this paper, we report an exploratory study of feature location process, consisting of two experiments in which developers were given unfamiliar systems and asked to complete six feature location tasks in two hours. Our study suggests that feature location process can be understood hierarchically at three levels of granularities: phase, pattern, and action. Furthermore, our study suggests 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 results open up new opportunities to feature location research, which could lead to better tool support and more rigorous feature location process.**

Keywords-feature location, human study, conceptual framework

I. INTRODUCTION

Developers often have to identify where and how a feature is implemented in the source code in order to fix bugs, introduce new features, and adapt or enhance existing features. This activity is referred to as feature location in the context of software engineering. Feature location has been recognized as one of the most common activities undertaken by software developers [1]. Due to the complexity of software systems and the cross-cutting concerns of features distributed in the source code, the process of feature location is time consuming and error-prone [2, 3].

To address this challenge, researchers have presented techniques to provide automated assistance in feature location tasks, using information retrieval [1, 4, 5], static analysis [5, 6], and dynamic analysis [1, 6]. These researchers have shown empirically that the proposed techniques can reduce the developer's effort in locating the implementation of features and improve the quality of feature-location results. In spite of the success of these feature-location techniques, feature location remains a human- and knowledge-intensive activity and it is risky to neglect human factors in the process [7]. Little research has been done on better understanding how developers perform feature location tasks. Thus, several important questions remain unanswered:

Q1. What actions (either physically or mentally) do developers commonly perform in the process of feature location?
Q2. Are there recurring patterns that reflect different searching strategies during feature location tasks? What affect the developer's choice of different patterns?
Q3. Are there distinct phases during feature-location tasks? What are the purposes of these phases? How do developers start, proceed, and finish the task?
Q4. Will the knowledge of feature-location phases, patterns and actions (if any) improve the performance of developers during feature location tasks?

To begin to answer these questions, we have conducted an exploratory study of feature location process. This study consists of two controlled 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-minute sessions. We analyzed the course of their completion of the assigned tasks in order 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-minute 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), respectively. 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 term of F-measure [2] of their feature location results) during feature location tasks with and without the knowledge of feature-location phases, patterns and actions.

The remainder of the paper is organized as follows. Section II reviews related work. Section III discusses the design of our exploratory study. Section IV reports the results and the quantitative analysis of the two experiments. Section V summarizes the insights and lessons learned from our exploratory study. Section VI discusses the external and internal threats to our study. Finally, Section VII concludes and outlines our future plan.

II. 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 information retrieval [1, 4, 5], static analysis [5, 6], dynamic analysis [6], or a hybrid of several analysis techniques [1]. The usefulness and effectiveness of these techniques have been evaluated and demonstrated empirically [8, 9].

In spite of the promising results of these (semi-)automatic feature location techniques, feature location remains a human- and knowledge-intensive activity [10]. This gives rise to the need for a more detailed understanding of how developers perform feature location tasks. Egyed et al. [10] 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, traceability granularity), and the effort and quality of recovering traceability links. Cuddeback et al. [11] presented a user study in which they investigated how human analysts examine candidate requirements traceability matrix produced by automatic techniques. However, little research has been done 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.

Ko et al. [3] conducted an exploratory study on how developers understand unfamiliar code during software maintenance tasks. They found that developers interleaved three activities, i.e. 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.

In this work, we focused on one specific type of program understanding tasks, i.e. feature location. Our study revealed a hierarchical conceptual framework for understanding feature location process. At the highest level, this conceptual framework consists of four distinct phases, Search, Extend, Validate, and Document. These phases are roughly at the similar level of abstraction to the three activities reported by Ko et al. [3]. We further investigated the actions directed towards 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.

III. EXPERIMENT DESIGN

Our study consists of two separate experiments with distinct objectives, i.e. the identification of distinct phases, recurring patterns and elementary actions in feature location process, and the evaluation of the effectiveness of the identified phases, patterns and actions during feature location tasks. Table 1 summarizes the subject systems, the participants, and the number of feature location tasks of these two experiments.

Table 1. Participants, subject systems and tasks

	Experiment 1		**Experiment 2**	
	JHotDraw	JPetStore	JGnash	Buddi
System Type	GUI Framework	Web Application	Finance Application	Finance Application
Tasks	3	3	3	3
Time (mins)	60	60	60	60
Version	7.4.1	1.0.0	3.4.2	4.1
Classes	700	73	511	253
Methods	7,256	385	3,975	1,863
LOC	72,911	2,314	43,957	18,755
Participants	18 graduate students and 2 full-time developers		18 third- and fourth-year undergraduate students	

A. Overview

Before each experiment, we introduced to the participants the background and relevant domain knowledge about the subject systems. Because we are interested in comparing 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 the 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-minute session to complete as many tasks and accurate as possible for each subject system. They had a 10-minute 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-minute session (Before session), the participants of group T_1 were given three feature location tasks on JGnash, while 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-minute break, the two groups swapped the subject systems and completed the other three tasks in the second 60-minute session (After session).

The participants were asked to fill in a post-experiment questionnaire to rate (based on a one to five scale) the perceived difficulty of their tasks, whether they have enough time to perform the tasks, and the perceived usefulness of feature-location knowledge they were taught. They were also asked to report (in free-form text) what they thought were the important factors that affect the feature location tasks.

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. The participants were asked to document their feature location results according to the given template, and submitted their results at the end of each experiment. We evaluated the feature location results of each participant with F-measure, i.e. the weighted harmonic mean of recall and precision [2]. 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.

B. Participaints

As our study consists of two separate experiments with distinct objectives, we recruited two entirely different groups of participants in two 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; and the remaining 10 had at least one year of experience on the design and development of various research tools. The two full-time developers had on average five years of industrial experience. Based on 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 use Eclipse in their daily work. Seven out of 20 participants reported that they have 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- and fourth-year undergraduate students from our school. Based on our pre-experiment survey, all of them used Eclipse "regularly" and reported on average 9.64 hours programming a week. 14 students described themselves as "above-average" Java expertise and the remaining four described themselves as "Java experts". Before this experiment, nine students only had experience with small projects less than 2,000 lines of code (LOC); six students had experience with projects of 2,000 to 10,000 LOC, and the remaining three had experience with projects of over 10,000 LOC. In this experiment, these 18 students were divided into two "equivalent" groups, based on their programming experience and capability. This allows us to perform a "fair" comparison of their performance on the same set of tasks.

C. Subject Sytems

Our study involves four open source Java systems as summarized in Table 1: 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:

- **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 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, while 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 [12]. 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 Spring framework. 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, while 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, currencies. The JGnash 3.4.2 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, etc. The Buddi 4.1 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 in order 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.

D. Tasks

In our study, each participant was given a sheet of paper describing their feature location tasks on a given subject system. Each task comes with a short feature name, a free-form textual description, and at least one typical usage scenario (see Table 2). For each task, the participants were requested to identify as many methods that they deem to be relevant to the given feature as possible. Developers had freedom to complete the assigned tasks in any order they prefer. They were also allowed to switch to another task if they find relevant information to that task while they are in 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 allows us to investigate variations in developer's strategies on different nature of tasks. Table 2 shows an example of each category. "Group/Ungroup Graphics Element" is an UI-related task for JHotDraw. It requires developers to make use of various UI-related hints, such as tooltips, graphic widgets. "Add New Transaction" is a database-related task for JGnash. It requires developers to effectively explore project packages and static dependencies. "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".

Table 2. Task examples

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

E. Development Environment

The participants were given the Eclipse 3.4.2 IDE 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 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 FLAT [13] or Re-Trace [14], because the primary objectives in this study are not to evaluate the effectiveness of such feature location tools.

F. Measures

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

Let T be a feature location tasks consisting of a set of features F. Let M be the set of methods in the implementation. Given a feature $f \in F$, let L^f_{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 locate the methods for all the features involved in our feature location tasks. This provides the ground truth (i.e. L^f_{actual}) for evaluating the participants' results.

Given $f \in F$, let $L^f_{reported}$ 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, i.e. $(L^f_{reported} \cap L^f_{actual})/L^f_{reported}$. Recall R_f is the percentage of actual traceability links reported, i.e. $(L^f_{reported} \cap L^f_{actual})/L^f_{actual}$. 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: $P_T = \sum_{f \epsilon F} P_f/|F|$, $R_T = \sum_{f \epsilon F} R_f/|F|$.

F-measure for a feature location task T is then computed as $f_b = (1+b^2)/(1/P_T + b^2/R_T)$. F-measure can be interpreted as a weighted average of the precision and recall. In this work, following the treatment in [2], we set b to 2, i.e. recall is considered twice as important as precision, because we deem that finding missing traceability links is more difficult than removing incorrect links.

IV. RESULTS

In this section, we describe and assess both quantitatively and qualitatively the empirical data that we collect in our study. Our analysis was based on 76 hours of full-screen videos of 38 developers' work on 12 feature-location tasks on four subject systems. First, we watched each developer's video to find the moments when the developer began, finished or switched the tasks. 71% of developers followed our instruction to record (by audio) 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, two authors cooperatively examined each task in detail, observing developer's actions and noting any interesting patterns and phases regarding how developers perform feature location tasks. This peer inspection allowed us to catch as much important information as possible. We have developed a simple tool called LogHelper to assist our analysis. This tool allowed us to create and maintain a log of each developer's work while we watched the videos, such as the developer's actions, their start and stop time, the order of these actions. It can also log our comments and inferences about developers' actions, such as the keyword they used in search, the types of dependencies they explored, the APIs they inspected, the location of breakpoints.

A. Actions

In the first experiment, we observed 17 initial types of physical actions and 16,203 physical actions in total. We

removed 3 types of uncommon actions, i.e. actions used less than 15 times, such as six "Reading irrelevant non-source files" actions. 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. Finally, we obtained 11 types of physical actions that developers commonly performed during feature location tasks in the first experiment. These are listed Table 3. In the first experiment, developers performed a median of 213 (±49) actions per task.

Table 3. Physical actions and their frequencies

Type	Information or activity	Freq
Read code	Source code, Comments, Javadoc, API specification	27.2%
Search program elements	Search text string or Java element, Inspect search result	8.5%
Explore static dependency	Type hierarchy, Declaration, Reference (call, access)	21.1%
Explore packages	Package explorer, Outline view	11.3%
Breakpoint operations	Toggle breakpoints, Disable/Enable breakpoints,	13.2%
Step program	Step into/out, Step over, Suspend	6.6%
Document results	Bookmark, Copy/Paste to a document, Writing notes	3.8%
Review task description	Task description	2.8%
Run program	Execute without breakpoints	2.8%
Edit code	Add/Remove comments, Print out messages	1.4%
Browse external resources	Internet; Dictionary	1.4%

As shown in Table 3, 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, document results, represent specific-purpose actions in feature location process. For example, document-results is 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 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 mental analysis that developers perform in mind. Although this list of mental actions is by no means complete, it allows us to better understand the developers' behavior during feature location tasks (see Section IV.C). 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

B. Phases

By studying logs of developers' actions, we observed four interleaved phases that fulfill four distinct purposes, i.e. Search, Extend, Validate, and Document. Figure 1 summarizes these four phases and their purposes in feature location process.

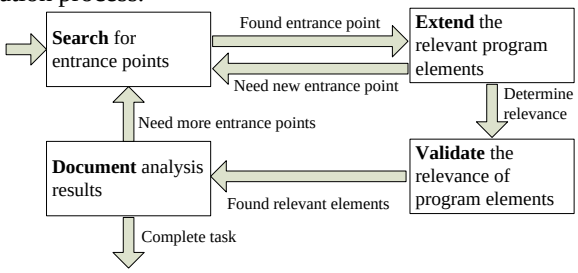

Figure 1. Four distinct phases of feature location process

Table 4. Heuristics for phase identification

Phase	Identification heuristics
Search	• The start of a new task • After documenting-results actions • Frequent search of program elements • Frequent static-dependency exploration • Stepping into the program • Run the program and observe • Browsing external resources
Extend	• Return to an element and explore its other information • Frequent static-dependency exploration • Stepping into the program
Validate	• Toggling/Enabling breakpoints • Quickly stepping over the program • Editing code and run the program • Printing out messages
Document	• Bookmarks • Copy/Paste elements to a document • Writing notes

Table 4 summarizes the heuristics that we examined in order to identify the phases of 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 dependency to validate the intent and behavior of an element. In our study, we combined the heuristics listed in Table 4 and other contextual information in order to determine the phases of developers' work during feature location tasks.

Due to the nature of feature location tasks, developers in our study always began by searching for entrance points, i.e. 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 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, 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.

Note that it was sometimes difficult to clearly determine the boundary between Search and Extend phase, 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 Search phase, while their actions were more specific during Extend phase. For example, during Extend phase, they often kept returning to a relevant element and explore 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 Search and Extend phase, developers in the Validate phase tended to quickly stepped over the program to the point they wanted to inspect.

Finally, developers documented the program elements that they deem relevant by bookmarking these elements, copying them into a document, or writing notes. If they believed that there are no more relevant program elements, they finished the task at hand. Otherwise they started a new round of Search, Extend, and Validate activities.

It is important to note that the four phases illustrated in Figure 1 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 is not relevant, he may go back directly from the Validate phase to the Search phase, without going through the Document phase.

C. Search Patterns

Based on the identified phases of 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, a forth of their time extending the relevant program elements, a sixth of their time validating the relevance of program elements, and a twelfth of their time documenting their analysis results.

This division of 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 Search and Extend phases. Third, we conducted experiments in 60-minute sessions. Developers often tried to improve their feature location results towards 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 Search phase than the other three phases.

Furthermore, the post-experiment questionnaires of the participants suggested that they consider finding the entrance points as fast and accurate as possible as one of the most important activities during feature location tasks. Our post-experiment analysis also suggested that the success of searching for entrance points has the big impact on the quality of feature location results, because it involves many speculation, errors, and useless operations.

The importance of Search phase and the challenges that developers may face in this phase motivated us to further study developers' actions within the Search phase. We identified three search patterns in the developers' work in the first experiment that reflects different strategies that developers adopted to find entrance points in Search phase.

1) IR-based Search Pattern

IR-based search pattern (see Figure 2) finds the entrance points for a feature location task by information retrieval 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.

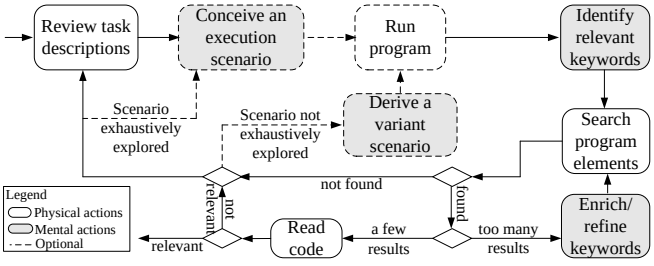

Figure 2. IR-based search pattern

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 in order 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 user interface (e.g. tooltip, 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 in order to find more cues.

2) Execution-based Search Pattern

Execution-based search pattern (see Figure 3) finds the entrance points by conceiving some execution scenarios and stepping the program. 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 the break-points based on the 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 need 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 in order to inspect different execution paths or different execution scenarios.

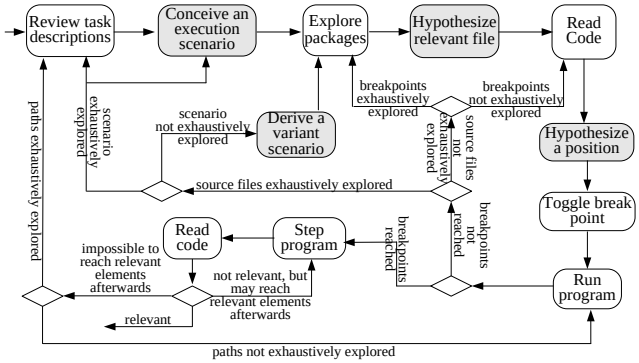

Figure 3. Execution-based search pattern

It is important to note that the roles that program execution plays in IR-based and execution-based patterns are completely different. The program execution in IR-based pattern provides an alternative way to identify keywords for program search. That is, it plays a supportive role to the primary technique, i.e. 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 pattern to find entrance points. Developers rely heavily on breakpoint operations and program stepping.

3) Exploration-based Search Pattern

Exploration-based search pattern (see Figure 4) finds the entrance points by exploring static program dependencies. A developer who adopts this pattern begins by exploring packages and hypothesizes 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) 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, type hierarchy) to reach more potentially relevant program elements.

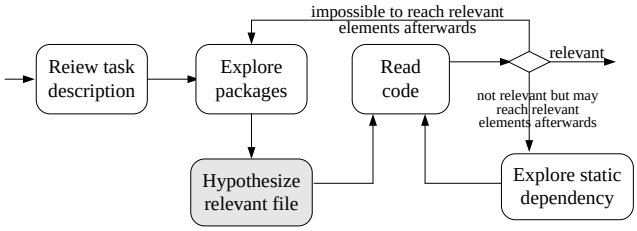

Figure 4. Exploration-based search pattern

D. The Quanlity of Feature Location Results

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 using 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% (±9.5%) F-measure.

In the second experiment, we taught the knowledge of the identified feature location phases, patterns and actions to two groups of junior developers. In the rest of this section, we report the impact of this knowledge on the performance of these junior developers on feature location tasks.

Table 5. Comparison of Result Quality by Subject Systems

System	Session	Recall (%)	Precision (%)	F-measure (%)
Buddi	Before (T_1)	56.3(±13.6)	74.9(±19.2)	58.4(±12.7)
	After (T_2)	72.5(±10.1)	62.4(±13.1)	69.1(±5.8)
JGnash	Before (T_2)	60.6(±13.3)	66.1(±17.2)	60.2(±11.4)
	After (T_1)	66.8(±9.7)	72.9(±10.1)	67.5(±7.4)
Overall	Before	58.4(±13.2)	70.5(±18.2)	59.3(±11.8)
	After	69.7(±10.0)	67.6(±12.6)	68.2(±6.5)

Table 5 summarizes the average recall, precision, and F-measure of the feature location results of these two groups of developers (T_1 and T_2). The "Before" and "After" rows of the subject systems represent the two experiment sessions before and after participants were taught the feature location phases, patterns and actions, respectively. The recall, precision and F-measure is reported as mean (± standard deviation).

Let us first examine the performance of the group T_1 on two subject systems. The "Before" row of the subject system Buddi and the "After" row of JGnash summarize the quality of the feature location results of the group T_1. Although JGnash is twice as big as Buddi, the group T_1 still achieved 10% increase in recall with only 2% decrease in precision. The overall quality (F-measure) of this group's results is improved by about 9%.

For the group T_2, the F-measure of their results on Buddi (the "After" row of Buddi) was improved by about 9%, in comparison with the F-measure of their results on JGnash (the "Before" row of JGnash). 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 on Buddi.

Let us now examine the performance of these junior developers on the same subject systems. Because we cannot compare the quality of the feature location results of the same participants on the same task, we comparatively study the result quality of two "equivalent" groups. As shown in Table 5, in comparison with the "Before" group (T_1 on Buddi), the "After" group (T_2 on Buddi) achieved on average 16% increase in recall with 13% decrease in precision. The F-measure of T_2 was improved about 11%. Similarly, the "After" group (T_1 on JGnash) achieved on average 6% increase in recall with 7% decrease in precision. The F-measure of T_1 was improved about 7%. Overall, participants achieved about 11% increase in recall with about 3% decrease in precision and about 9% increase in F-measure.

We also investigated the quality of each participant's results individually. 16 out of 18 participants had better F-measures (ranging from 57.0% to 68.4%). Among these 16 participants, eight had both better recalls and precisions, while the other eight had better recalls but worse precision. We further investigated the performance of participants with different program experience. Based on the participants' pre-experiment survey, we identified 8 experienced developers and 10 less experienced. 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. Furthermore, the reduced standard deviation, evident in the quality of participants' result by task (see Table 5), also indicates that the gap between each individual participant was also reduced.

In a summary, we observed certain improvements in the quality of the participants' feature location results. However, due to the short period of time participants had to absorb the knowledge of feature location phases, patterns and actions that they 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.

The participants were not aware of how we measure the quality of their results. They indicated in the post-experiment interview that they did not intentionally boost recall by simply including more program elements. Thus, we believe there exist certain correlation between the improvement of the participants' feature location results and the relevant knowledge of feature location they were taught. However, more studies are necessary to further expose this correlation.

E. The Choices of Search Patterns

Finally, let us examine how often each search pattern (identified in Section IV.C) has been adopted during feature location tasks and what their potential impact is on the quality of feature location results. Table 6 summarizes the statistics we collected 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.

Table 6 Pattern usage and its impact on result quality

Pattern	Session	Usage	F-measure (%)
IR	Before	10	55.1
	After	10	64.3
Exploration	Before	3	60.5
	After	4	72.9
Execution	Before	5	67.1
	After	4	73.6

As expected, IR-based pattern has been adopted most frequently (by 10 participants) in the second experiment, while 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 was 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 18 participants. For the second participant who adopted 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 find 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 (see Table 6), 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.

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 patterns is the primary factor that affects the quality of their feature location results.

V. DISCUSSION

Conducting this exploratory study has given us some interesting insights into the process of feature location. We discuss them in this section.

A. A Conceptual Framework for Understanding Feature Location Process

Our study suggests a hierarchical conceptual framework for understanding feature location process. As shown in Figure 5, this conceptual framework consists of a collection of phases, patterns and actions. A phase consists of collections of concrete actions directed towards 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 actins (see Section IV.A) that are important to understand the feature location process. We identified four phases with distinct purposes in feature location process, i.e. 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 Search phase.

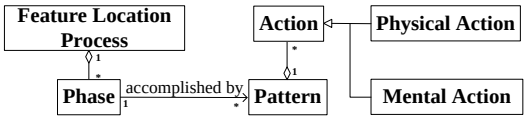

Figure 2. Conceptual framework for understanding feature location process

This conceptual framework provides an organized way to describe and understand feature location process. It indicates that feature location process engages several level 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 process.

This conceptual framework also allows researchers to start focusing on the human factors in the feature location process. In this work, we reported 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 tradeoff of different tactics, so that developers could make more informed decision in different situations. Although our results are by no means conclusive, they suggested that the proposed conceptual framework is promising in improving developers' effectiveness on feature location tasks.

B. Intelligent, Interactive, Online Feature Location Tools

Helping developers explore and understand code has been an important challenge in software engineering research. Several tools have been built to support developers' work in different phases of the proposed conceptual framework, such as Robillard and Murphy's work on concern graph [15] and relevant tools [16,17,18]. Our conceptual framework for feature location process could provide a foundation for better integration of existing tools.

Furthermore, as discussed in Section IV.C, different search 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 pattern, static dependencies and program navigation in exploration-based pattern. However, these patterns also seek other supportive sources of information and tactics. This finding gives rise to the need for better integration of different sources of information and different searching, debugging and exploring tactics during feature location tasks.

Finally, our detailed understanding of feature location process 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 developers' feature location tasks.

For example, this wizard could monitor and analyze the developer's actions on the fly. Unlike existing tools [16] 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 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 IV.E, developers may tend to have their consistent preference for the 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. Based on this model, the wizard could then offer personalized support for 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 developer's work, especially in Search and Extend phase.

As shown in Section IV.C, 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→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.

VI. 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 of in a real-world context. Although we recruited developers with industrial experience, adopted four real-world subject systems, 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.

Participants in our study were asked to work independently and on the 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 length of experiment session (60 minutes) 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 2.94 (±1.30). Based on the screen videos of the participants' work, we observed that about 84% of the participants completed their assigned tasks within 44–53 minutes. However, this 60-minute time constraints may pressure the participants to complete all of the tasks as fast as possible and thus affect their actions during the experiments.

Many findings in this study were based on our subjective interpretations of the full-screen videos of developers' work. The complexity of 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. 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.

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.

VII. CONCLUSTION AND FUTURE WORK

In this paper, we reported an exploratory study of feature location process with two experiments, involving 38 developers, 4 real-world subject systems, and 12 feature location tasks. Based on the empirical results of this study, we proposed a conceptual framework for understanding feature location process, which consists of a collection of phases, patterns and actions. This framework allows us to further investigate the human factors in 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 the future, we plan to extend our conceptual framework with a more complete set of patterns, especially for other phases (e.g., Extend) of feature location process that have not been investigated in detail in this study. Furthermore, we are interested in developing an intelligent, interactive, online feature location wizard, based on our findings in this work.

REFERENCES

[1] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Trans. Softw. Eng.*, 33(6): 420-432, 2007.

[2] J. Hayes, A. Dekhtyar, S. Sundaram. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Trans. Softw. Eng.*, 32(1): 4-19, 2006.

[3] A. Ko, B. Myers, M. Coblenz, H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.*, 32(12): 971-987, 2006.

[4] A. Marcus, J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. *ICSE*, pp. 125-135, 2003.

[5] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. *ACM Trans. Softw. Eng.*, 15(2): 195-226, 2006.

[6] T. Eisenbarth, R. Koschke, D. Simon. Locating Features in Source Code. *IEEE Trans. Softw. Eng.*, 29(3): 210-224, 2003.

[7] J. Hayes, A. Dekhtyar. Humans in the Traceability Loop: Can't Live With 'Em, Can't Live Without 'Em. *Proceedings of the 3rd Workshop on Traceability in Emerging Forms of Software Engineering*, 2005.

[8] A. De Lucia, R. Oliveto, G. Tortora. Assessing IR-based traceability recovery tools through controlled experiments. *Empir Software Eng*., 14(1):57–92. 2009.

[9] B. Cleary, C. Exton, J. Buckley. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empir Software Eng*.,14(1):93-130, 2009.

[10] A. Egyed, F. Graf, P. Gr ünbacher. Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments. *RE*, pp. 221-230, 2010.

[11] D. Cuddeback, A. Dekhtyar, J. Hayes. Automated Requirements Traceability: the Study of Human Analysts. RE, pp. 231-240, 2010.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] T. Savage, M. Revelle, D. Poshyvanyk. FLAT3: Feature Location and Textual Tracing Tool. *ICSE*, pp. 255-258, 2010.

[14] A. De Lucla, F. Fasano, R. Oliveto, G. Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Trans. Softw. Eng.*, 16(4): 13-50 , 2007.

[15] Martin P. Robillard, Gail C. Murphy. Representing concerns in source code. PhD Thesis, Univ. of British Columbia, 2007.

[16] Martin P. Robillard, Gail C. Murphy. Concern graphs: Finding and Describing Concerns Using Structural Program Dependencies. *ICSE* pp. 406-416, 2002.

[17] Martin P. Robillard, Gail C. Murphy. Automatically Inferring Concern Code from Program Investigation Activities. *ASE*, pp. 225-234, 2003.

[18] Martin P. Robillard. Automatic generation of suggestions for program investigation. *ESEC/FSE*, pp. 11-20, 2005.