Supporting Comprehension of Unfamiliar Programs by Modeling an Expert’s Perception

by

Naveen Kulkarni, Vasudeva Varma

in

36th International Conference on Software Engineering, Hyderabad

Hyderabad, India

Report No: IIIT/TR/2015/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
May 2015
ABSTRACT

Developers need to understand many Software Engineering (SE) artifacts while making changes to the code. In such cases, developers use cues extensively to establish relevance of an information with the task. Their familiarity with different kind of cues will help them in comprehending a program. But, developers face information overload because (a) there are many cues and (b) they might be unfamiliar with artifacts. So, we propose a novel approach to overcome information overload problem by modeling developer’s perceived value of information based on cues. In this preliminary study, we validate one such model for common comprehension tasks. We also apply this model to summarize source code. An evaluation of the generated summaries resulted in 83% similarity with summaries recorded by developers. The promising results encourages us to create a repository of perception models that can later aid complex SE tasks.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Programming Environments—Programmer workbench; I.2.5 [Artificial Intelligence]: Programming Languages and Software—Expert system tools and techniques

General Terms
Human Factors, Measurement, Experimentation

Keywords
Program Comprehension, Summarization, Information Scent

1. INTRODUCTION

Developers spend significant time to extend the life of a mature software system by fixing bugs or modifying programs [12]. During this task, developers constantly face the challenge of comprehending unfamiliar programs. Though there are variety of tools and techniques, a considerable effort is required to identify parts of systems relevant to their current task [10]. Through tools that recommend artifacts [4] are helpful, developers still have to investigate each artifact to find task specific information. LoToza et al. [11] refer to this as ‘fact finding’ where facts are derived from code and other SE artifacts such as bug reports or commit logs. They suggest that any approach helping developers to comprehend should consider all such facts. Several other observations on program comprehension effectiveness also suggest the need for methodical approaches [18, 9]. We intend to model how an experienced developer considers different facts of a program for a given task. This will be helpful for those developers who are less experienced and unfamiliar with the program.

Developers seek specific information depending on the task. It is found that developers exhibit foraging behavior while seeking information [10, 18]. In such cases, Foraging Theory suggests the use of an information scent – a perceived value of information based on cues to assist searching. Further investigation by Lawrance et al. [13] also established that developers hypotheses before foraging and use information scent to verify or refute them. They modeled a developer’s foraging behavior by observing their navigation for a given task. However, we believe that there are many cues (not just proximal) that a developer uses. For example, when a developer investigates a method it is not just the method-name cue, but non proximal cues such as comments (API and inline), parameter names and access restriction are also used. However, we posit that in spite of using the cues developers face information overload.

Our research goal is to aid developers while comprehending an unfamiliar program. Each SE artifact offers many direct and indirect cues for comprehension. For example, in a program we find direct cues in lexicons and natural language comments. Whereas, marker interfaces, conditions, typecasts and exceptions suggest indirect cues. In addition, tools can offer visual cues such as formatting, hover text and lexical coloring. Further, cues can also be induced by developer actions such as searching, introducing deliberate error or by program analysis. So, we like to study.

A. What cues are used by developers to comprehend an unfamiliar program?
Comprehension techniques also influence a cue selection. When faced with an unfamiliar program, developers read code, its documentation, or simply execute it [2]. Typically, reading code is a preferred option, since it factually represents the true state of a system [12]. On the other hand, though good insights can be drawn from system execution traces, understanding the relationship between observed behavior and unfamiliar code is difficult. Similarly, a system documentation rarely reflects the rich mental model with which the programs are constructed. Hence, we restrict our further discussion to cues used while reading code and also interested in applying the results of A to study.

B. How can we use the knowledge of cues to help a novice developer in making a program investigation faster?

In section 2, we propose a novel approach to aid program comprehension based on the cues. In Section 3, we apply the prior knowledge on cues to synthesize program summaries, followed by a discussion in Section 4 and related work in Section 5.

2. DEVELOPER’S PERCEPTION MODEL

When reading a program, developers often use a combination of cues depending on the task. For example, Ko et al. [10] observed that to understand a source file’s intent, developers first related file names to the functionality, opened the file if relevant and then inspected method names. However, if it were to be a debugging task then developer would choose a call hierarchy rather than file name cue to reach an intended method. Unfortunately, not much is known about the correlation between cues and a SE task. We believe this correlation is representative of a developer’s perception.

We hypothesize that all developers follow same information scents while comprehending an unfamiliar code (without advance tools). If this were to be true, then all developers should develop similar perception. Since perceptions are based on cues, understanding cues used by developers and their preferences can aid comprehension task (study objective A). We refer to this as “modeling developer’s perception”. To investigate this further we define a perception model for a simple comprehension task based on our experience and literature study. We evaluate this model by validating against human responses. We also show that such perception models significantly improve the quality of automated program comprehension.

2.1 Model for Comprehending an Unfamiliar Program

Our view on comprehending an unfamiliar program includes not just the developer’s intentions [6], but also the program’s evolution and auxiliary information like bug posts or discussions. In this context, we define a rudimentary model as shown in Table 1. Our model suggests that developers investigate a concept based on cues such as lexicons, constraints, validations and comments (in-line, commit). Similarly, cues for comprehending a procedure, feature and module are listed. However, not much is known about the developers preferences towards cues. So, we build our initial model with an assumption that developers would equally consider each cue during their investigation. Though it is intuitive that developers will use intrinsic cues like natural language lexicons, our intention was to explore the extent to which they use extrinsic cues (not readily available as part of a program).

2.2 Model Validation

We validated our proposed model using a case-study. The validation setup included 15 developers investigating the source code of jEdit\(^1\), version 4.3.2, an open source text editor written in Java. This project was chosen because (a) none of the developers were conversant with the jEdit code, (b) all developers were aware of a general text editor features and (c) jEdit is an active projects in Sourceforge with access to all the SE artifacts. We defined two tasks, first to comprehend a section of a procedure and second, to comprehend a concept in a procedure.

Task 1: What is the purpose of loadCaretInfo in EditPane.java?

Task 2: How pop-up menu items are added through GUIUtilities.java?

Developers who participated in our study had 1 to 10 years of Java programming experience. We had three categories with five developers in each category. Category 1 had practiced SE for greater than 30 months, Category 2 had practiced SE for less than 30 months, and Category 3 consisted of developers with good programming skills but limited SE practice. During the evaluation, we provided all the participants a condensed archive of carefully selected SE artifacts (bug history, commit history, discussions, test cases, API descriptions and links to all repositories) related to the prescribed tasks. Later, each participant was asked to record a summary of their understanding for Task 1 and Task 2. They were also asked to record time taken to complete and the cues (from Table 2) they referred for each task. On an average developers took 12.4 minutes for Task 1 and 8.5 minutes for Task 2.

\(^1\)http://jedit.org

| Table 1: Developer’s perception model for program investigation |
|---------------------------------|----------|---------|---------|----------|
| Comprehension task related to   | Concept  | Procedure| Feature  | Module    |
| Developer Intension            | Code lexicon (explicit cues) | ✓       | ✓       |          |
|                                | In-line Code Comments         | ✓       |          |          |
|                                | Control flow (implicit cues)  | ✓       |          |          |
|                                | Test cases (unit)             | ✓       |          |          |
| Boilerplate                    | API Documentation             | ✓       | ✓       | ✓        |
|                                | License                       |          | ✓       |          |
| Program Evolution              | Commit Comments               | ✓       | ✓       |          |
|                                | Version history changes       | ✓       | ✓       |          |
| Auxiliary                      | Structure (Dependency etc.)   | ✓       | ✓       |          |
|                                | Properties (Modularity etc.)  | ✓       | ✓       |          |
|                                | Bug Reports                   | ✓       | ✓       |          |
|                                | Discussions                   | ✓       | ✓       |          |
Table 2: Cue distribution across SE artifacts based recorded summary

<table>
<thead>
<tr>
<th></th>
<th>Cate-</th>
<th>Code</th>
<th>Control</th>
<th>In-line</th>
<th>Test</th>
<th>API</th>
<th>Commit</th>
<th>Version</th>
<th>Bug</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gory</td>
<td>lexicon</td>
<td>flow</td>
<td>Comment</td>
<td>Cases</td>
<td>document</td>
<td>Comment</td>
<td>History</td>
<td>report</td>
<td></td>
</tr>
<tr>
<td>Task 1</td>
<td></td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Task 2</td>
<td></td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tf-idf Task 1</td>
<td>0.346 (1)</td>
<td>0.170 (3)</td>
<td>0.136 (5)</td>
<td>0.030 (7)</td>
<td>0.306 (2)</td>
<td>0.151 (4)</td>
<td>0</td>
<td>0</td>
<td>0.055 (6)</td>
<td></td>
</tr>
<tr>
<td>tf-idf Task 2</td>
<td>0.483 (1)</td>
<td>0.357 (3)</td>
<td>0.475 (2)</td>
<td>0.066 (6)</td>
<td>0.252 (4)</td>
<td>0.180 (5)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

We performed analysis on a contingency table (refer Table 2), where rows are developer category and columns are cues. We used Intraclass Correlation$^2$ (ICC) measurement to find the degree of agreement between each developer category using a similar cue pattern. We found that there was an almost perfect agreement (ICC >0.8) among all developer categories on cues referred (Task 1, ICC$^1_{1,2} = 0.89$, ICC$^2_{2,3} = 0.83$, ICC$^1_{1,3} = 0.75$; Task 2, ICC$^1_{1,2} = 0.94$, ICC$^2_{2,3} = 0.94$, ICC$^1_{1,3} = 0.94$). So, our hypothesis that all developers follow same information scent holds good. We found from our Pearson Chi-Square test that developers did not refer to all cues in an artifact during both the tasks (p-value <0.01). This indicates that developers either have strong preferences towards a certain cue or there is a trade-off for better cues. For example, most of the developers did not find it necessary to include conditions or checks in their summary though they used code lexicon cues extensively.

To understand if cues referred by developers are task dependent, we used term frequency and inverse document frequency ($tf$-$idf$), a common information retrieval measure. The $tf$-$idf$ value is used as a representative quantity to indicate the importance of a cue (extracted as key words from recorded summaries) used by the developers. Previous research suggests that word frequency and inter-word correlation can successfully predict a developer’s behavior (choice of cues in this case) for a comprehension task [16]. Hence, the summaries recorded by developers represent their perception. We calculated the $tf$-$idf$ values for all the terms in the recorded summaries against all possible cues extracted from SE artifacts (provided during evaluation). The average values and the observed cue priority are presented in Table 2. We found significant differences in cue priority for both tasks (p-value <0.01) suggesting that cues referred by developers are tasks dependent.

### 2.3 Threats to Validity

To provide a meaningful program summary, it is essential to understand a developer’s information foraging behavior. In doing so, we have proposed a model and evaluated it. Following are the threats to validity that can affect our result.

a) **Internal validity:** A major threat to our evaluation are the number of participants in our study and the limited number of tasks assigned to them. Instrumentation effects may have resulted due to the evaluation materials employed. During evaluation, a condensed archive was provided to developers to ensure they do not diverge from the assigned task. Also, all developers used a simple text editor to accomplish the tasks. Because of these we might have missed observing new cues. Other internal validity threats can be due to the learning effect experienced by the developers. This effect was mitigated by our evaluation design: All developers were given a task at a predefined time. They had to complete the task before taking up the next task. The next task was assigned a week after completing the first task.

b) **External validity:** Though our evaluation include fewer developers, we ensured they are randomly selected and with a wide range of experience. In our choice of developers we also considered their SE process knowledge and grouped them accordingly. The software system chosen for evaluation is fairly large and well maintained. This makes the comprehension task realistic. However, our prescribed tasks required developers to investigate only a single program file, where as in practice developers are required to investigate more than one program file. Hence, further studies with different types of software systems and comprehension tasks are required to confirm or refute the obtained results.

c) **Construct validity:** To ensure an unbiased evaluation, the hypothesis of our study was not revealed to the participants. We chose common comprehension tasks to ensure our results are generalizable for code inspection. But, the program comprehension tasks are also undertaken for tracing defects, preparing test cases and enhancements. We believe our choice of statistical measures – ICC and $tf$-$idf$ are good representative when comparing cues used by developers. However, the term matching only reflects the existence of a term in a document (and its importance). This limits the evaluation because of vocabulary problem discussed in the next section.

### 2.4 Impediments for Perception Modeling

During our study we interviewed all three category developers and identified that they faced difficulty in comprehension due to following two reasons.

---

$^2$Traditionally, ICC has been used to measure the reliability of a subject’s rating when a subject is rated by a pool of raters.
1. Multiple identifiers used by developers for a concept leads to vocabulary problem [5]. These are predominantly due to casual practices like inconsistent trimming of identifiers (like isPriNumber for checking prime number) and using atomic identifiers (like verbs, remove or update). During Task 1, developers identified that they had difficulty in understanding textArea, line 401. They initiated a search and found a cue at line 279, EditPane.java. Nevertheless, they had to initiate a new comprehension task even before finishing the current task. There are attempts to overcome this problem within source code [1]. However, resolving vocabulary problem across SE artifacts is challenging.

2. Cues misrepresent content due to scattered functionality. This issue is caused if developers do not follow single responsibility principle or ensure modularity while modifying. We observed this in Task 1, as developers remarked that they did not get any hint of bug fix at line 456 (EditPane.java) from cues like method name or API comment. This was also reported by Ko et al. [10], where developers investigate a different part of software due to misleading identifiers during their study.

These impediments can potentially impact perception modeling. In addition, we observed that expressiveness in programming language also influences program comprehension. For example, loosely typed Java 1.4 collection package required developers to comprehend additional type checking boilerplate code for restricting a specific type of objects. But with type safe feature added in Java 1.5 onwards it provided better expressibility of developer’s intention. Hence, an expressive programming language provides better cues.

3. AUTOMATED PROGRAM SUMMARY

Summarization is suggested to be an effective technique to aid program comprehension [7, 17, 6]. In addition, comprehension will be better if program authors describe their intentions using natural language [11, 18, 10]. For example, more comments (API and inline) in source code are always appreciated. Also, it is suggested that full lexical terms used for identifiers lead to better comprehension [5]. So, a program’s comprehensibility could improve if a summary is generated from its natural language lexicons. However, extracting all the lexicons will cause information overload and also results in same summary for all the program comprehension task types. We used our proposed model (Section 2.1) to generate precise yet useful summaries for investigative tasks (Study objective B).

3.1 Investigative Code Summarizer

Code (a section of a program) summaries should provide developers a good overview and avoid repeated investigation of same code during modification. Developers explicitly search for specific information and ignore the rest even if it is evident [18]. For example, a developer investigating a concept might ignore a related bug fix only to revisit it later after finding an evidence (from bug history or commit comments). However, a experienced developer spend more time investigating gathering information and rarely revisit. So, we use our perception model described in Table 1 to reproduce this behavior of experienced developer. The code summary hence generated can prevent reinvestigation.

Our study suggest source code lexicon as an important cue while investigating unfamiliar program. But, meaningful construction of summary using lexicon will be difficult due to interleaving of code fragments. For example, in our case study we observed that the concept ‘buffer’ was interleaved (lines 330 to 336, 348, 354, 357 and 360 to 366) in EditPane.java. Hence, to overcome the interleaving issue we use Concept Lattice, a proven technique used to represent conceptual hierarchies [8]. Using the inherent ordering of the lattice structure, we extract code lexicons to synthesize a summary. Further, summary was also constructed by considering lexicons from access methods in a program, first line of natural language text from comments, bug reports and discussions related to the program. Term matching was done to remove sentences with duplicate concepts.

3.2 Evaluation and Results

To evaluate the synthesized summaries we use the previously recorded summaries of Task 1 and 2. Table 3 shows two such recorded and a synthesized summaries for Task 1. To evaluate the relevance of a summary we used a content evaluation technique. This type of evaluation emphasizes on conceptual similarity between automated and manual (control) summaries. For content based evaluation, text analysis community prescribes Pyramid method [15]. In addition, we use unique terms as measure of summary length.

The results of evaluation are shown in Table 4. We found equal number of unique terms in the summaries recorded by all three developer categories. This suggests that length of summary is directly related to number of cues used by the developers during a program comprehension task. Hence, we conclude that developers have similar perception model despite different background. The pyramid scores indicate a high degree of content similarity between manual and synthesized summaries. This is a significant improvement when compared to other proposed summarization techniques [6].

These result are encouraging as it suggests that our initial rudimentary model made summaries that are relevant for a general investigation task. However, despite similar number of terms and comparable pyramid scores the our automated summaries had a high readability index. We believe this can be overcome by carefully considering the cue priorities and better natural language generation approaches.

4. DISCUSSION

Studies on developer’s foraging behavior can lead to interesting solutions and the findings from our study can greatly improve their productivity. We were interested in learning the cues that the developers use in practice during program

<table>
<thead>
<tr>
<th>Table 4: Evaluation of automated summary; *Avg. without stop words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>Manual</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Task 1</td>
</tr>
<tr>
<td>Task 2</td>
</tr>
<tr>
<td>Task 3</td>
</tr>
</tbody>
</table>
comprehension (objective A). When our proposed cue model was applied for Task 1 and Task 2, it did provide summaries similar to the human recorded ones for common program comprehension tasks. It is interesting to find that despite difference all the developers used similar cues for a given task. Further, our study also complements the finding from [3, 11, 13] that there exists a strong correlation between the tasks and the cue priorities. Further, we found developers ignoring cues that were evident. Hence, our initial assumption that a developer would consider all the cues was invalidated. We also found that developers not just rely on explicit cues but also look for implicit cues. For example, the developers who participated in our study suggested that indentation and grouping of lines of code too provide them a clue.

During the evaluation of all the summaries recorded by the developers we observed that an abstractive rather than an extractive approach was followed by all. An extractive summary will include text from the artifacts where as an abstractive summary is synthesized using natural language generation techniques on some semantic representation of the content. Also, the average summary length was roughly 10% of lines of code they investigated (refer Table 4). This is an interesting finding since it provides us an input on the length of summary. But, there is a need for a further study to establish if there exists any correlation between the length of the summary and a program comprehension task.

5. RELATED WORK

This paper is a contribution to the extensive literature on aiding developers to improve their productivity. Specifically, our focus was to use both implicit and explicit information available in all SE artifacts to construct a natural language summary related to a program comprehension task. While researchers have proposed many automated approaches to aid developers, summarizing a program has gained significance recently. In this section, we shall survey these new approaches.

Information foraging theories applied to model a web navigation behavior by Pirolli [16] has been a source of inspiration for our work. Applying this theory to SE maintenance tasks such as program comprehension has been extensively studied [12, 10, 18, 13]. In particular, we build our approach based on the interesting results reported by LaToza et al. [11]. They suggest that expert developers can easily navigate in a complex code because they seek precise evidence of relevance to the task when faced with the decision to investigate an unfamiliar code. The use of information scent for a maintenance task broadly discussed by Lawrence et al. [13] also closely relates to our approach. They describe a model for program navigation (foraging path) where as we explicitly modeled information scent (cues) in various SE artifacts. Similarly, Crosby et al. [3] highlight how typical indicators in a program called “Beacons” facilitate code recognition through their experiments. They have reported that experienced developers tend to rely on beacons more often than novice developers.

Recent studies suggest several interesting approaches to summarize a program. Sridhara et al. [19] have suggested approaches to synthesize natural language summaries for Java programs by extracting textual content from control flow graphs. Similarly, Moreno et al. [14] have proposed an approach using automatically identified stereotypes within a Java Class. The summaries generated from both these approaches have a fixed template and primarily describe the structure of a program. In contrast, we believe that a program summary should be task specific. Hence, our approach differs from the above two in synthesizing task specific natural language summaries for Java programs by extracting textual content from various SE artifacts.

Haiduc et al. [6] have used a text retrieval (TR) technique (as applied for a natural language document) to synthesize program summaries. Despite employing a TR technique their approach is limited to extracting structural summaries of a program. On the other hand, we consider all SE artifacts associated with a program. As all TR techniques are statistical in nature, they scale well and can handle various type of artifacts. So, in future we intend to use them in our approach. We consider Rastkar et al. [17] work similar to ours in generating an abstractive summary for a program. Their approach uses ontologies and extracted identifiers to synthesize abstract summaries. But, these summaries are limited to a cross cutting concern of a program.

6. CONCLUSION

A program comprehension is a complex task that requires a developer to refer multiple SE artifacts. This overwhelms
a developer making a comprehension task challenging. To overcome such an information overload, experienced developers often rely on cues that are explicit or implicit in an artifact. These cues are found to be effective in assisting developers to create a mental model of a program. On the other hand, novice developers are found to ignore them. However, in spite of such cues, developers find comprehending an unfamiliar program hard.

Providing an automated program summary in natural language specific to a developer’s task can greatly improve their productivity. We attempted to learn the cues used by the expert developers for a general program investigation task to develop our proposed perception model. From our evaluation we found that the knowledge about cues can improve the quality of a program summary. The resulting summaries were contextual and significantly resembled to those recorded by the human expert developers. Also, the generated summaries were precise and avoided superfluous information, a common issue in summary generation.

The encouraging results from our preliminary study motivates us to further investigate the role of cues in automated program summarization. We intend to extend our study to include complex tasks and explore the role of induced cues as well. We also plan to build a set of perception models for common program comprehension tasks. Also, very little has been done in exploring interesting synergies between program comprehension and Text analysis. Considering the increasing number of diverse SE artifacts, a text analysis technique can prove to be effective by providing a fast and a scalable approach for developer productivity tools.

7. ACKNOWLEDGMENT

We like to thank all the developers from Infosys Labs, Hyderabad, India and IIIT-H graduate students for participating in our study. We also like to thank all the anonymous reviews for their valuable feedback.

8. REFERENCES


