In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 160 - 172

The Role of Source Code within Program Summaries describing Maintenance Activities

Pamela O'Shea and Chris Exton

University of Limerick, Castletroy, Limerick, Ireland,
{pamela.oshea, chris.exton}@ul.ie,
WWW home page: http://www.b4step.ul.ie

Abstract. This paper investigates the use and type of source code employed during program summaries which describe software maintenance tasks. The data consists of eighty-eight program summaries extracted from online developer mailing lists. The summaries were categorised into three themes, description of problems, modifications and modification requests. Each theme was subdivided into five task types, adaptive, corrective, emergency, perfective and preventive. A subset of three categories from a content analysis schema have been isolated for this investigation, the three source code categories cover descriptions of single lines of code, code excerpts and blocks of code. The use of these three categories were examined between the theme groupings as well as within the themes, that is, differences between the task types. The results were not as frequent as expected. However a significant difference was found between the adaptive and perfective modification request summaries.

1 Introduction

Program maintenance and program comprehension are rarely discussed in isolation, for example, Pressman cites the following: *"Program maintainability and program understanding are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain"* [16]. It is this *understanding* which provided the motivation for a larger study, a subset of which is reported upon here where the role of source code during program summaries was investigated.

The wider research question under investigation may be stated as follows, *"How can the experienced Java programmer be supported during software maintenance ?"*. In terms of this paper, a subset research question may be stated as, *"What role does the source code play during maintenance for experienced Java programmers ?"*. The terms *"software maintenance"* and *"experienced programmer"* will now be defined and discussed in turn.

Software maintenance has been defined many times by researchers, including Lientz and Swanson in 1978, where it was stated that *''maintenance and enhancement are generally defined as activities which keep systems operational and meet user needs"* [10]. A more recent definition may be taken from the software engineering institute's glossary at carnegie-mellon which defines maintenance as *"the cost associated with modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment"* [3].

These and other definitions show that maintenance always consists of *"activities"* which involve the modification of the existing source code in some way. This paper presents an investigation into the abstractions used and described by programmers when performing such modifications to well established open source, object oriented, Java systems.

For the purposes of this study, the term *"experienced programmer"* may be placed in the context of online open source projects where accepted peer developers contribute source code to the repositories. Open source projects were chosen for investigation [1] for a number of reasons, firstly as Capiluppi et al stated, *"open source software provides a good opportunity for observing software products in various phases of their evolution, growth and maturity"* [2].

Secondly, a large amount of publically accessible resources are associated with well established projects, such as CVS (Concurrent Versions System), issue-tracking, communication changes (e.g. mailing lists and newsgroups) and online documentation [6].

Thirdly, experienced programmers may be observed through the archives of developer mailing lists. For well established projects, at least two separate mailing lists exist, one for end-users and another for project developers. It is the latter, developer mailing lists which have been examined in this study. The developer lists have been chosen from the Apache foundation's Jakarta site for open source Java projects [1].

The final reason for choosing open source data is that accounts closely representing program summaries may be found within the online mailing lists, where developers discuss problems and design issues, modifications made, etc. Program summaries are accounts written by a programmer to summarise their understanding of the program under investigation. Such summary accounts have previously been gathered within a laboratory setting by Pennington [14] and Good [7] for analysis, where programmers were asked to provide a written summary of a given procedural program. Building upon Pennington's work, Good applied a developed schema to categorise the documented summaries. For example, categories within the schema included, data flow, control, function, action, operation, state-high, state-low, meta, elaborate, incomplete and unclear (due to space constraints, the reader is referred to [7] for full definitions and examples, however their definitions do not affect the understanding of the results presented here). One possible analysis could be to classify the summaries as either control flow or data flow, etc. by examining the frequency of occurrence of each category within the program summaries.

For the research question, a schema was developed during an initial study, based upon Good's work, to describe the views and abstractions found within the Java program summaries. An overview of this schema will be presented in Section 2.1. The schema was then documented and a coding manual written to allow other researchers to use the schema. The schema was then applied to the data gathered from the online mailing lists.

It is this developed schema which was designed to encompass both the *views* of the software system (those categories developed by Good with a number of additions for the application to object oriented summaries) and the *abstractions*. It was found that the programmer's accounts could be classified into *what* the programmer was describing

[1] Approved by the University of Limerick Ethics Committee (Application Number: 03/52).

and *how* it was being described. The *views* are used to describe *how* the programmer was describing the item in question, for example, if the programmer was describing a variable, was it the control flow or data flow, etc., of the variable being described ? In contrast, the *abstraction* was *what* the programmer was describing, e.g. was the programmer speaking of a package, class, object, variable, line of code, etc ?

Section 2 will present the decisions made during the design of the study, including the schema design in Section 2.1, procedure in Section 2.3, sample size in Section 2.2, reliability in Section 2.4, the themes and task types used to categorise the summaries in Section 2.5, and Section 2.6 outlines the hypothesis for this paper. Section 3 will present the results where the summaries are considered as part of their assigned task types and themes.

2 The Study

As stated in the introduction, the source of the data was chosen to include program summaries extracted from online, open source, mailing lists. As no single study with observed participants is without flaws, the following disadvantages were considered when comparing this methodology to a laboratory study: 1. Cannot easily monitor any of the user's interactions with the debugger, 2. Cannot verify the correctness of the participants' statements as it is too time consuming to read the open source project source code in order to verify the participants' statements, 3. Contact would be required to be made with the participants off-list in order to verify their exact level of expertise, 4. Cannot control the target audience of the summary i.e. whether the participant was writing a summary for a peer programmer in the same project/group or whether the summary was intended for an unfamiliar programmer or novice etc.

The following advantages were deemed important enough to choose this data gathering method over the controlled laboratory method or even in-situ/action-research studies where the presence of an experimenter is required: 1.Increased possibility of achieving strong ecological validity i.e. the participants are unaware of being monitored and are forming the summaries within their own working environment using familiar tools, 2. Larger sample sizes are possible, i.e. 88 program summaries were gathered in total, 36 of which provided the *modification* results presented here (Table 1), 3. The programs being described are both large and real and not manufactured by the experimenter in any way, 4. The tasks performed are real tasks, 5. Regular contributors to the developer mailing list are most likely to be experienced as they are accepted by their peers.

2.1 Schema Development

To encompass all the abstractions found within a typical Java program summary, the following *abstraction* schema structure was created with three levels, high, middle and low. The higher abstractions are used when the programmer describes abstractions above the program level in their descriptions, i.e. at the software architecture/design level. The middle abstractions are used when the programmer was describing the program/modification within the program level. The lower abstractions are used when the

programmer is describing the program/modification near the source code level and Java virtual machine (jvm) level.

The higher abstractions (system/architecture level) include the following abstraction descriptions: package (pac), program (prog), thread (thr), component (com), interface (int), abstract class (abs), class (cla) and object (obj). For example, if a programmer refers to a class as part of the program summary, then the cla category may be applied to that description.

The middle abstractions (within program level) include the following abstraction descriptions: external (ext), feature (fea), algorithm (alg), program unit (pru), constructor (con), method (met), variable (var), and process (pro). The less verbose terms will now be discussed. The external abstraction here refers to any output of the program, for example, the program output itself or a file. A feature classification was used when the programmer was describing a particular feature of the program and makes references to it as such. The program unit was used to refer to a small unit of executable code i.e. an action performed by the program.

The lower abstractions (source code and jvm) include the following abstraction descriptions: block (blo), code excerpt (cex), line of code (loc) and java virtual machine (jvm). The block abstraction is equivalent to *"chunking"*, where the programmer refers to a block of code as an item, for example, a *try-catch-finally* block which is often found in exception handling parts of Java programs. A code excerpt (cex) is when the source code was directly included within the summary or attached to the message (the size is also recorded without comments). The Java virtual machine (jvm) refers to any output of errors (exceptions, stack traces), garbage collection or any other jvm activity. The program unit (pru) referred to in the middle abstraction does not belong here as it describes executable code, the code here must be shown and described statically.

2.2 Sample Size

The sample size of 88 was chosen to satisfy the content analysis coding, where Krippendorff's sample size table was used to determine the appropriate figure [9].

The number 88 was derived from the initial study involving 58 summaries producing 845 segments. Inferences were not to be made regarding the two least frequently occurring categories in the *"views"*: *"Bucket"* and *"Unclear"*, which left the *"State"* category as the next least frequently occurring category (occurring 36 times in total). At the 0.05 significance level, 44 was determined from the table[2]. This recommended sample size was doubled to 88 for two reasons. Firstly, to improve reliability and secondly to record more summaries for each of the task types (adaptive, corrective, emergency, perfective and preventive) under each of the summary themes (modification, modification request and description of problem).

2.3 Procedure

The data was gathered by searching each developer mailing list at the Jakarta Java site [1] for the keyword "summary". The search was performed in reverse chronological order (beginning at the date of the initial study in December 2003 and working backwards)

[2] Verified through personal correspondence with the author Krippendorff.

with a maximum cap of fifteen summaries on each list. A cap of fifteen was placed as there were many lists available and to allow for a selection of summaries from various projects/domains. It was not possible to automate/randomise the data gathering process as each message which was returned with the keyword required reading by the author to ensure it was indeed a program summary. Those messages which were not program summaries were discarded. A summary may be defined as an entire message or part of a message which describes a program for the benefit of another reader. Where a message is any single post made to the developer list.

Once each of the program summaries were gathered from the developer mailing lists [1], the described schema was then applied to each summary. This process employed the content analysis method where a number of predefined categories are applied to a text. The presence of such categories and their relationships can then be examined allowing inferences to be made, which in our case are inferences regarding the experienced programmer's abstraction usage.

The first step involves, splitting the summaries into segments, with each segment consisting of a subject and a predicate. Each segment was then examined and categorised into one of the 22 abstraction categories. Each categorisation is mutually exclusive, e.g. a segment classified as describing an *object* cannot also be classified as describing as *class*.

For example, the following is a segmented summary from the study (each forward slash represents the end of segment and the beginning of a new segment):

```
Pass a string into the unescapeHtml() method / that contains a hex
entity / (i.e. &#xB7; instead of &#183;) / and you will get a
NumberFormatException. / The offending code is in Entity.java, line
690. / It should check whether the character after the # is 'x' /
and if so, / prefix it with '0' / and call Integer.decode().intv() /
(or some other hex converting function). / Hex entities are valid
HTML /
(http://www.htmlhelp.com/reference/html40/entities/latin1.html) /
so this should be supported.
```

Before the application of the schema takes place, the structural variables are recorded, which consist of the following: message identification number, thread name, theme, task type, author, message subject, date and time. Next, each segment within the summary is given an identification number and the content variables are applied to each segment. The content variables consist of those abstraction categories already described in Section 2.1.

2.4 Reliability

Krippendorff's Alpha [9] and Cohen's Kappa [5] are two common methods of reporting reliability in content analysis studies.

The Kappa statistic was calculated by comparing the results of 8 independently coded summaries from one researcher and measuring the agreement with the results from the coding performed by the first author. The Kappa was found to be 0.8818 using the documented coding assumptions. A further improvement of 0.9449 was found after an agreement was made regarding the *"meta"* category for programmer comments. However, the first Kappa result must be used as protocol.

2.5 Themes and Task Types

Table 1 shows the distribution of the 88 summaries within each task type.

	Adp.	Cor.	Em.	Perf.	Prev.	Total
Mod.	1	18	0	14	3	36
	2.78%	50%	0%	38.89%	8.33%	100%
MR.	3	0	0	9	0	12
	25%	0%	0%	75%	0%	100%
Dop.	2	28	1	4	5	40
	5%	70%	2.5%	10%	12.5%	100%

Table 1. Distribution of task types within summary themes (88 Summaries)

Each summary was found to fit into three distinguishable themes or aims. That is, the theme or purpose of the program summary could be easily classified as either a description of how a *"modification"* was performed, or an account of a *"modification request"* or a *"description of a problem"*. Each theme was further sub-categorised into one of five task types. The themes are defined as follows:

Themes

- Modifications (Mod): source code for part or all of the solution must be included in the program summary, either embedded, attached or the location referenced.
- Modification Request (MR): is only a request for a feature, no implementation discussion or solution must appear or be attached/referenced within the message.
- Description of Problem (DOP): the problem and/or the possible solution is described, but no solution code must appear or be attached/referenced within the message.

Task Types The five task types are composed of the four types defined in the *IEEE Software Maintenance Standard* [8], *"adaptive"*, *"corrective"*, *"emergency"* and *"perfective"*, while the fifth task type is one that is often referred to in the software maintenance literature as *"preventive"*.

Through the decades, all the literature has agreed on three of the maintenance tasks, those being, *adaptive*, *corrective* and *perfective*. For example, Swanson in the 1970's [18], Martin and Osbourne in the 1980's [12], and the updated IEEE software maintenance standard in the 1990's [8]. It is the *"other"* category which has varied widely, however, *"preventive"* has been consistently appearing most frequently in the literature as the *"other"*, e.g. [16] and [15].

While *adaptive*, *corrective* and *perfective*, may be the most consistently agreed upon task types, the definitions may be seen to vary throughout the literature. This fact is also highlighted recently by Chapin [4].

Chapin et al make a clear and useful distinction between task type definitions derived from empirical work (activity-based) and theoretical (intention-based) based definitions. For the purposes of this empirical study, the summaries were clearly divisible into the five task types using the following definitions.

- Adaptive: *"accommodation of changes to data inputs and files and to hardware and system software"* [10]. Examples found include, modifications made to support distributed systems were also considered adaptive.
- Corrective: *"emergency fixes, routine debugging"* [10]. Examples of this category were quite frequent.
- Emergency: *"Unscheduled corrective maintenance performed to keep a system operational"* [8]. Examples found include, modifications made to correct discovered/reported security bugs.
- Perfective: *"user enhancements, improved documentations, recoding for computational efficiency"* [10]. Examples found include, making modifications to support programmers enhancements/needed features, as well as performance optimisations.
- Preventive: *"work performed on a system in an effort to prevent an error or malfunction from occurring"*, cited in [15] (originally by [13]). Examples found included the following:
  - preventing problems before they occur, e.g. skewed files
  - improving design for extensibility
  - performing modifications to support practices, such as calling *"super"*
  - designing/recoding to avoid situations such as out of memory errors
  - performing an operation to be in-line with correct procedures e.g. initialisation of variables
  - making modifications to prevent of deadlocks
  - correcting access control on variables, e.g. *"transient"*

Referring to Table 1, it is interesting to compare the distribution of task types within the *"modifications"* theme, where the *"corrective"* task types occurred 50% of the time, followed by *"perfective"* at 38.89%. These results are comparable with Vans, von Mayrhauser and Somlo [19] where it was stated that *"Corrective maintenance is a frequent activity during software evolution"*. A more recent study performed by Schach et al [17] showed a 53.4% and 56.7% usage for *corrective* type maintenance at the module and change-log level respectively. The results are also similar to the 50% found in this study. Interestingly, their *perfective* results are similar to the 38.89% usage found here also (36.4% and 39% for the module and change-log level respectively).

In contrast, Pfleeger stated that *perfective* task types take 50% of the total maintenance effort, while the *corrective* tasks are shown to consume as little as 21% [15]. The results of this study also conflict with Lientz and Swanson [11], where *corrective* maintenance was found to have a frequency of 17.4%, while *perfective* maintenance was found to take 60.3% of the maintenance effort.

2.6 The Hypothesis

As stated in the introduction, the authors wish to examine the role of source code within the program summaries. As a result, the hypothesis under investigation in this paper

may be stated as:

"Source code is the most important abstraction used within the program summaries for all three themes".

That is, if the author is describing where and how the modification was made, or where they are experiencing problems with implementation, or requesting a modification, a greater emphasis is expected to be placed on the role of source code the locations within the program where the modifications were made. Consequently the null hypothesis may be stated as *"The role of source code is not more important than any other abstraction for all three themes"*.

3 Results

All results are first examined for differences between the three themes, followed by differences within the themes, i.e. between the task types. Significance was set at the .05 level, while the anova test was used to examine differences within the groups. If the anova was significant, a post-hoc scheffe was then performed to find out which groups differed.

Section 3.1 will now present the results where the three source code abstraction categories (block, code excerpt and line of code) were merged into one source code category (termed actual_code) and compared to other types of abstraction usage (termed code_other). Section 3.2 presents the results of the three individual code categories (block, code excerpt and line of code) against the rest of the abstractions (termed code_other).

3.1 Code Usage Vs Other Abstractions

Within the three themes, a comparison of the code usage against the usage of other abstractions did not yield any significant results as shown in Figure 2, where p = .265. As a result, the null hypothesis could not be rejected.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	.166	2	.083	1.329	.265
Within Groups	73.554	1179	.062		
Total	73.720	1181			

Table 2. All Themes: Code Vs Other: Anova Results

However, when investigating for differences between the task types within the three themes, only one significant difference was found between the adaptive and perfective

tasks within the modification request theme as shown in Figure 3. The null hypothesis could be rejected for the modification requests tasks as p = .016. As shown in Table 1, only two tasks types were found within the modification request theme, adaptive and perfective. As a result, a post-hoc test was not required. Figure 4 shows the counts for both these tasks, it can be seen that the adaptive task uses source code (actual_code) approximately twice more than expected, while the perfective tasks use source code less than expected. However it should be noted that the lack of source code references within the perfective tasks may be due to the fact that perfective activities themselves often involve changing documentation and as a result, source code references would be reduced.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	.514	1	.514	5.927	.016
Within Groups	11.196	129	.087		
Total	11.710	130			

Table 3. Modification Requests: Code Vs Other: Anova Results

		Adp.	Perf.	Total
actual_code	Count	6	7	13
	Expected Count	2.7	10.3	13.0
code_other	Count	21	97	118
	Expected Count	24.3	93.7	118.0
Total	Count	27	104	131
	Expected Count	27.0	104.0	131.0

Table 4. Modification Requests: Code Vs Other: Frequency Results

3.2 Types of Code Usage Vs Other Abstractions

This section presents the results of the three individual code categories, unlike the previous section where all three were merged as one, as a result the null hypothesis may be stated more verbosely as follows:

The use of code excerpts, lines of code and blocks of code categories were not anymore or less important than the use of other abstractions within each of the three themes.

Between the Themes Within the three themes, a comparison of the types of code usage against the other abstractions did not deviate significantly from the expected count. The anova results are shown in Figure 5 with p = .478, as a result, the null hypothesis could not be rejected at the theme level.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	.110	2	.055	.739	.478
Within Groups	87.481	1179	.074		
Total	87.591	1181			

Table 5. All Themes: Code with Other: Anova Results

Within the Themes No significant difference was found within the modifications or description of problem themes. However, an investigation within the modification request theme showed a significant difference. Since only two task types exist within this theme, a post-hoc was not required as the difference can only be between the adaptive and perfective summaries. Figure 6 shows the anova results with p = .003, allowing the null hypothesis to be rejected for the two summaries within the modification request theme. Figure 7 shows the crosstab breakdown with expected counts for each code category and the other abstractions (code_other) where it can be seen that the significant difference is with the code excerpt category. Adaptive summaries used code excerpts almost six times more than expected, while perfective summaries used code excerpts almost three times less than expected.

Figure 8 shows the types of code excerpts, where three of the eight were java (*_j), the remaining were xml (*_x). Figure 9 shows the lengths of the java excerpts (valid column) which averaged at six lines.

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	1.059	1	1.059	9.310	.003
Within Groups	14.667	129	.114		
Total	15.725	130			

Table 6. Modification Requests: Code with Other: Anova Results

		Adp.	Perf.	Total
blo	Count	0	1	1
	Expected Count	.2	.8	1.0
cex	Count	6	2	8
	Expected Count	1.6	6.4	8.0
code_other	Count	21	97	118
	Expected Count	24.3	93.7	118.0
loc	Count	0	4	4
	Expected Count	.8	3.2	4.0
Total	Count	27	104	131
	Expected Count	27.0	104.0	131.0

Table 7. Modification Requests: Code with Other Crosstab

	Frequency	Percent	Valid Percent	Cumulative Percent
Valid f_cex_j	2	25.0	25.0	25.0
is_cex_x	3	37.5	37.5	62.5
loc_cex_j	1	12.5	12.5	75.0
loc_cex_x	2	25.0	25.0	100.0
Total	8	100.0	100.0	

Table 8. Modification Requests: Types of Code Excerpts

	Frequency	Percent	Valid Percent	Cumulative Percent
Valid 3	1	33.3	33.3	33.3
4	1	33.33	33.3	66.7
11	1	33.3	33.3	100.0
Total	3	100.0	100.0	

Table 9. Modification Requests: Sizes of Code Excerpts

4 Conclusion and Future Work

This paper described a schema for object oriented Java program summaries. A classification consisting of three themes and five task types was also described allowing future investigations to research differences between these themes and task types. The results presented were a small subset of the categories gathered, blocks of code, code excerpts and lines of code. The role of source code was surprisingly less frequent than expected by the author.

The null hypothesis could not be rejected for the majority of cases tested, that is, between the themes and within the themes (at the task level). However, for the modification requests the null hypothesis could be rejected for the two tasks within its theme. The adaptive and perfective tasks showed a significant difference in their use of code excerpts. Adaptive summaries placed more emphasis on code excerpts, while perfective summaries place less emphasis on code excerpts.

One reason for the lack of source code references may be due to the fact that the programmers were selective about which source code was discussed. That is, the programmer has already refined their search offline and is writing a post mortem summary. However, it is still surprising that the modification theme did not show greater amounts of code excerpts as a means for displaying the authors work.

Future work includes examining which abstractions are more important than source code and how these can help the design of software visualisation tools.

Acknowledgements

The authors have been supported by the Science Foundation Ireland Investigator Programme, B4-STEP (Building a Bi-Directional Bridge Between Software ThEory and Practice).

We would also like to thank the anonymous reviews for their helpful comments and suggestions.

References

1. Apache Jakarta, Open Source Java Project Page Available at: http://jakarta.apache.org
2. Capiluppi, A. et al.: Structural Evolution of an Open Source System: A Case Study. Proceedings of the IEEE International Workshop on Program Comprehension (2004) 172–182
3. Carnegie Mellon Software Engineering Institute Glossary. Available at: http://www.sei.cmu.edu/str/indexes/glossary/software-maintenance.html
4. Chapin, N. et al.: Types of software evolution and software maintenance. Journal of Software Maintenance, 13:1, John Wiley & Sons, (2001) 3–30
5. Cohen, J.: A coefficient of Agreement for Nominal Scales. Educational and Psychological Measurement, 20 (1960) 37–46
6. Cubranic, D., Murphy, G. C.: Hipikat: Recommending Pertinent Software Development Artifacts. Proceedings of the IEEE International Conference on Software Engineering (2003) 408–418

7. Good, J.: Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Comprehension. Ph.D. Thesis, University of Edinburgh (1999)
8. IEEE standard for Software Maintenance. IEEE Std 1219-1998, (1998)
9. Krippendorff K.: Content Analysis: An Introduction to Its Methodology. Sage Publications, Second Edition, ISBN: 0-7619-1545-1 (2004) 122
10. Lientz, B. P. et al.: Characteristics of application software maintenance. Communications of the ACM, 21:6 (1978) 466–471
11. Lientz, B. P., Swanson, E. B.: Characteristics of Application Software Maintenance. Communications of the ACM, 21:6 (1978) 446-471
12. Martin, R. J., Osborne, W. M.: Guidance on software maintenance. National Bureau of Standards Special Publication 500. (1983)
13. Miller, J.: Techniques of Program and System Maintenance. Winthrop Publishers (1981)
14. Pennington, N.: Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology, 19, (1987) 295–341
15. Pleeger, S. L.: Software Engineering: the Production of Quality Software. Macmillan, 2nd Edition, ISBN 0-02-395115-X. (1991) 420
16. Pressman, R. S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, 5th Edition, ISBN 0-07-709677-0. (2000) 849
17. Schach, S. R. et al.: Determining the Distribution of Maintenance Categories: Survey versus Measurement. Empirical Software Engineering, 8, (2003), 351–365
18. Swanson, E. B.: The Dimension of Maintenance. Proceedings of the Second International Conference on Software Engineering. (1976) 492–497
19. Vans, A. M. et al.: Program Understanding Behavior during Corrective Maintenance of Large-Scale Software. International Journal of Human-Computer Studies, 51:1, Academic Press, (1999) 31–70