Investigating Patterns and Task Type Correlations in Open Source Mailing Lists for Programmer Comprehension

Pamela O'Shea and Chris Exton
University of Limerick
Ireland
{Pamela.OShea, Chris.Exton}@ul.ie

Keywords: POP-II, POP-III, POP-V, Content Analysis, Program Comprehension, Abstraction, Software Visualisation.

Abstract

This paper describes an investigation using data from a number of open source mailing lists. A methodology is described that details the procedure used to extract program summaries posted to mailing lists by experienced programmers as well as the type of analysis involved. The data are analyzed for patterns and correlations between task type and the program comprehension scheme categories used. When compared to laboratory style empirical studies, an advantage is gained allowing stronger ecological valid insights into programmer comprehension.

Introduction

The study described in this paper was performed in order to examine the patterns of abstraction levels used by experienced programmers when describing a program. The data was also examined for evidence of task type correlations, i.e. whether the task type affected the type of abstractions used. From such results, requirements may be gathered for a supportive Software Visualisation (SV) tool allowing appropriate abstractions of the software system to be developed based on the gathered data. For example, it was found that data flow and function type abstractions were repeatedly used together during the program summaries. Such evidence supports the need to allow the user to easily switch from the data flow view in the software visualisation tool to higher abstractions.

Gathering such data using empirical studies requires the setting up of an environment that is unfamiliar to the programmer. Monitoring equipment for talk-aloud, video and keystrokes are often used. The tasks given are often artificially created by the experimenter. Even in more immersive approaches where the experimenter works within the participants' workplace forces the presence of an unfamiliar person. In both these cases the environment is unnatural and the participant is aware of being monitored.

Using content analysis of Open Source mailing lists provides a wealth of information while overcoming many of these issues. Used on its own or in conjunction with other studies a rich data set can be obtained. Open Source lists deal with a real, large and more often than not, industry standard program. Such projects attract regular contributions from programming enthusiasts who are both experienced and knowledgeable of their domain and language. The tasks being performed are real and require solving within a reasonable amount of time. The environment is not manufactured in any way by the researcher. Content analysis in this manner is unobtrusive as the participant is not even aware of the experimenter or monitoring of any kind.

As already stated, the investigation described here is motivated by the need to gather requirements for a SV tool. For the purposes of clarity, SV will now be defined and discussed before the study is detailed.

Software Visualisation was concisely described by (Ball and Eick 1996) as the following, "*Software is intangible, having no physical shape or size ... Software Visualization tools use graphical techniques to make software visible through the display of programs, program artifacts, and program behaviour*".

Typical real-world software systems, are quite complex and difficult to comprehend. An in-depth knowledge of such a system takes significant investments of time. As a result, a large cognitive load is placed on the maintainer upon seeing the code for the first time. Software Visualisation aims to help the programmer carry such a cognitive burden. Consequently, SV tools are bound to the results from program comprehension studies to provide direction in achieving this goal.

This paper describes a study that analyzed programmers' summaries found in Open Source Java mailing lists. The benefits of which are described in next Section. The motivation behind the study is discussed further along with the study itself. Good's program comprehension scheme (Good 1999) was used for the analysis of the gathered program summaries. This scheme, is discussed further in the Procedure Section. It is particularly suitable in helping us to find the appropriate abstractions needed by programmers. Allowing us to appropriately measure the abstractions used. The Procedure Section also describes how the lists were chosen and how the summaries were selected from these lists without a bias of one type of summary over another. A description is provided of how the program comprehension scheme was applied to these summaries.

Following this, the results, discussion and conclusions are detailed.

Advantages of Examining Online Mailing List Data

Software Visualisation tools are often evaluated by comparing the levels of program comprehension achieved by its users during controlled experiments.

However, one could argue that experimental controls associated with many of these studies impede on the ecological validity of the obtained results. This is due to the fact that the programmer is in an unfamiliar/unnatural environment and will quite possibly function and behave differently than under more normal circumstances, i.e. empirical experimental controls have potential to lack ecological validity.

On the other hand, what other options do we have when dealing with professional programmers in an industrial setting? Immersive approaches such as action research are possibilities. For instance to perform research within the company over a longer period of time and gather data from within the programmers' own environment. Which technique is used is a matter of suitability for the type of data that is required.

The method we have used here is to gather the results from the wild in an unobtrusive manner. The raw data is gathered from online mailing lists and analyzed, by doing so, ecological validity is maintained as the programmer is unaware of the observational study.

Software Visualisation Abstractions

Figure 1 shows a software system (source code files shown on the left) which can be examined at many levels of abstraction. The highest abstraction is the software architecture view, further down the abstractions are the relationship view, data flow view and control flow view. The dots in between each of these views in Figure 1 shows that many have yet to be defined/discovered through empirical evidence gathered from experienced programmers. At the bottom of the abstractions is the source code itself.

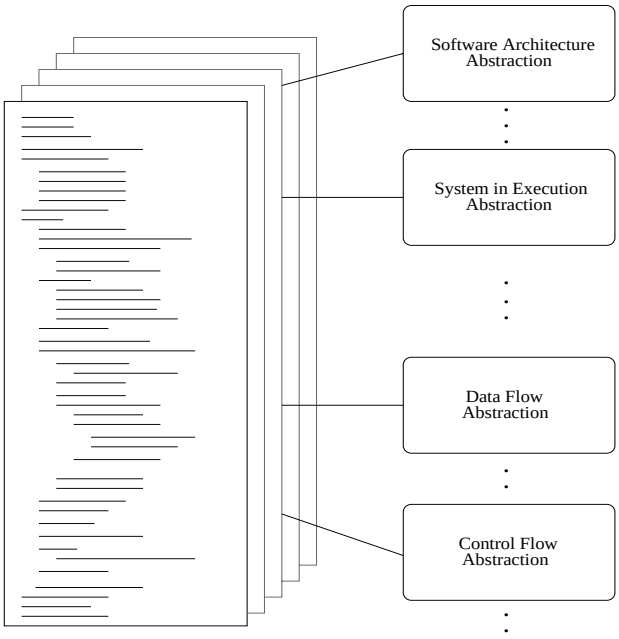

Figure 1: Presenting Abstractions of a Software System

The programmer accumulates much of this abstraction information from many varying sources during comprehension (e.g. from source code, documentation, run-time analysis of the system, etc.)

Appropriate visual representations of the software can be used to abstract the system into many views and present them to the user for further investigation. Each abstraction view is synchronised with each active view. That is, an event in one abstraction view will cause all the other visualisation windows to be updated.

The tool is to be integrated with an existing industry standard development environment. It will also be closely integrated with the debugger to support interactive user queries. The supported language is to be an Object Oriented language environment. Java was chosen as it is widely used in industry and provides various monitoring capabilities as part of its implementation.

For our purpose Open Source Java mailing lists provide a wealth of information. Well known Open Source projects were examined and experienced/regular contributors were chosen as part of the study. Much of the software is used in industry and attracts industry standard contributions. It is program summaries from such lists that have been analyzed in order to provide the requirements for the SV tool.

Our objective can be stated in the form of a research question as follows: *"Which level of abstraction or levels of abstractions of the software system are most important to the experienced programmer during maintenance?"*

For the purposes of this study, we are most interested in finding what patterns of abstractions are most used by the programmers in their summary accounts. These results will provide feedback for the abstractions that need to be implemented and closely integrated in the tool.

Procedure

Open Source Java mailing lists were identified[1] and summaries/programmers were chosen equally from each list (data permitting).

[1] Although there are many Open Source Java projects, our list was chosen from: http://jakarta.apache.org/

Once the lists were chosen, the posts were listed from the most recent to the oldest. Threads were searched for messages that contained program summaries. A program summary was identified using the keyword *"summary"* as the search criteria. In the majority of cases, the summaries were contained within submitted bug reports and patches. All program summaries were recorded until fifteen[2] summaries were collected from each list.

Two types of variables were gathered from the program summaries. Both structural variables and content variables were gathered as was done by (Gold and Auslander 1999).

Structural Variables consisted of the following:

- **Mailing List Name**: The name of the project that the developer mailing list belongs to.
- **Thread Title**: The subject of thread being discussed.
- **ID**: An identification number was assigned to each message and programmer
- **Reason for Summary**: For example, if the programmer was requesting help, giving help, describing a bug or solution etc.
- **Summary Length**: The total number of segments in the summary.
- **Code Inserts**: Whether the message contained a source code insert. This meant that the source code for one or more lines of code was counted.
- **Documentation References**: Whether the summary contained references to Javadoc, API, Comments etc.
- **Architectural References**: Whether the summary described the architecture of the system in some way. Examples include the use of the following keywords: *(inherits, subclass, superclass, interface, abstract, extends, protected, private, public, static*, etc.). As well as any comments made in general about the design, for example, ``*this is not designed to handle...*".
- **Error References**: Whether the message included any error messages. Examples include the following: Full/Partial Stack Traces, Exceptions, Compiler Errors, Runtime Errors, etc.
- **File Name References**: Whether the summary names the Java files by name, i.e. any *.java references.

Content variables consisted of the categories from the program comprehension scheme (Good 1999). There are eleven categories plus one rarely used additional category donated by a minus sign ("-") for segments that could not be categorized in any way. Examples of such statements were: ``*The patch is attached to this message."* These content variables will now be enumerated along with examples from the study.

- **Function (F):** The overall aim of the program is described succinctly. Example: *"is intended to represent exceptions thrown explicitly by JS throw statement"*
- **Actions (A):** Events occurring in the program which are described at a lower level than function. Example: "*we need to try each of the IP addresses*".
- **Operation (O)**: Small scale events which occur in the program, such as tests, assignments, etc. Example: *"from SMTPHandler doRCPT()"*.
- **Data (D):** Inputs and Outputs to programs, data flow through programs, and descriptions of data objects and data states. Example: *"it remains as zero"*.

[2] This number was chosen in an effort to keep the number of summaries from each list evenly distributed. It is a typical number of participants that would have been used in an empirical study.

- **State-High (SH):** High-level definition of state. An event is described at a more abstract level than state-low. Example: *"If the DNS lookup fails"*.
- **State-Low (SL):** Lower version of state-high. State-Low usually relates to a test condition being met, or not met, and upon which an operation depends. Example: *"Consider, too, that this feature works only if STATE is "ERROR""*.
- **Control (C):** Information having to do with program control structures and with sequencing, e.g. recursion, calls to subprograms, stopping conditions. Example: *"But instead it exits"*.
- **Unclear (U):** Statements which cannot be coded because their meaning is ambiguous. Example: *"and the height is recorded"*. It is not clear here whether *"recorded"* means *"printed"*, *"added to a list"*, *"assigned to a variable"*, etc.
- **Incomplete (I):** Statements which cannot be coded because they are incomplete. Examples include unfinished sentences.
- **Meta (M):** Statements about the participant's own reasoning processes. Example: *"As I see it this is because the code catches ArrayIndexOutOfBoundsException"*.
- **Elaborate (E):** Further information about a process/event/data object which has already been described. Example:*"(which would be mx1.mail.yahoo.com)"*.

During the study, the summaries were split into segments before coding, e.g. the following statement was split into segments as shown below.

"This is the class that goes through a file and gets the data to be graphed, in this case port numbers.".

"/ This is the class / that goes through a file / and gets the data / to be graphed, / in this case port numbers ./".

Each segment is examined and categorized into one of eleven categories. The results were calculated as a percentage of the number of occurrences of the category type to the total number of segments in the summary.

Each of the eighteen programmers remained anonymous and is given an identification number. During the study, the programmers will only be referred to by their identification number (i.e. Numbers 1-18). Multiple summaries by these programmers are examined for both usage patterns and correlations.

The individual programmers were selected from five different mailing lists. This was to eliminate bias on the type of descriptions used to describe the programs. For example, programmers working on the *Lucene* search engine project would be expected to produce more data type descriptions than programmers working on the *James* (Java Apache Mail Enterprise Server) project. By having a broad range of projects where programmers were selected from, such biases are reduced. The mailing lists will now be enumerated with a short description of the domain each belongs to.

- **Slide Project** (4 programmers: 1, 2, 3, and 4)

  "The Slide project main module is a Content Management and Integration System, which can be seen as a low-level content management framework" (Slide Project).

- **Lucene Project** (4 programmers: 5, 6, 7, and 8)

  "Jakarta Lucene is a high-performance, full-featured text search engine written entirely in Java" (Lucene Project).

- **James Project** (3 programmers: 9, 10 and 11)

  "The Java Apache Mail Enterprise Server (a.k.a. Apache James) is a 100\% pure Java SMTP and POP3 Mail server and NNTP News server" (James Project).

- **BSF Project** (2 programmers: 12 and 13)

``Bean Scripting Framework (BSF) is a set of Java classes which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages". (BSF Project).

- **jEdit Project** (5 programmers: 14, 15, 16, 17 and 18)

  "jEdit is a ... programmer's text editor that has been in development for over 5 years" (jEdit Project).

Results

Three types of patterns were recorded: function, data and control type categories. These are enumerated below. The notation used is with an arrow operator. For example, the forward arrow (d->**f**) shows that data flow descriptions were often followed by functional type descriptions. The function (f) category is shown in bold to highlight that the pattern was found when examining functional type statements. The three highest patterns are described for each pattern type.

The summaries were categorized into five task types, "Code Explanations" (the author was attempting to explain either their own code or somebody else's code to a poster), "Help Requests" (the author was asking for help by describing their problem), "Providing Help" (the author was attempting to help the poster by providing solutions), "Proposed Changes" (the author is detailing their intentions to change or update or even add a feature to the program) and "Bug Descriptions"(the author has found a bug and is describing a report of how it can be replicated).

The task type is also reported along with the three most popular categories used throughout such summaries. Additionally, the most frequently used categories for the first, second and third opening segments of the summary were also recorded, in an effort to find if there is a difference in the level of abstraction used by the programmer when first discussing the problems of different task types, i.e. where does the programmer being the investigation?

Function Type Patterns

The most frequently used function-type pattern was the "d->**f**" pattern. The "**f-**>d" pattern was the next most frequently used. Both of these were used by eight of the eighteen programmers. This was followed by the "**f-**>sh" pattern which was used by seven programmers.

Data Type Patterns

The "f->**d**" pattern was the most frequently used by nine of the eighteen programmers. Seven programmers used the "**d-**>f" pattern, while six programmers used the "**d-**>**d**" pattern.

Control Type Patterns

The "**c-**>d" pattern occurred most frequently, used by five of the eighteen programmers. The "d->**c**" pattern was found to be the next most employed pattern by four programmers, followed by the "**c-**>**c**" pattern as the third most frequently used.

Task Type: "Code Explanations"

The highest three categories found were function, data and state-high. The top three categories used for the first, second and third segments were function, function and function respectively.

Task Type: "Help Requests"

The highest three categories recorded were control, function and data. The top three categories used for the first, second and third segments were data, function and control respectively.

Task Type: "Providing Help"

The highest three found were function, data and state-high. The top three categories used for the first, second and third segments were function, stat-high and elaborate respectively.

Task Type: "Proposed Changes"

The three highest categories used here were function, meta and data. The top three categories used for the first, second and third segments were function, data and control respectively.

Task Type: "Bug Descriptions"

The three highest categories used overall were data, function and state-high. The top three categories used for the first, second and third segments were function, data and meta respectively.

Discussion

The results recorded in relation to the patterns of category usage have an impact on how the abstraction levels in the SV tool should be integrated. These will now be discussed in turn.

The function category switched to data most often. The next highest interaction was with the state-high category. From the SV tool perspective, the data flow views should be integrated with the higher abstractions in order to facilitate such switching.

The data category interacted with the function category most frequently. This was followed by interactions with other data type statements. The need to easily switch between function and data and vice versa within the SV tool is highlighted again here.

Surprisingly the control category interacted the most with the data category. This was followed by interactions with other control type statements. Showing that data and control type views are not as exclusive as they are currently treated within SV tools. Support is found here for integration with the debugger, in such a way, variable values can be found for example from the control flow view at any time.

The results from the examination of task types and the number of categories used found less differences than expected. However, there are some interesting results (e.g. for the "Help Request" task type), which will now be discussed.

The "Code Explanations" task type employed the same three highest categories as the "Providing Help" task type and the "Bug Description" task type. Interestingly, the categories were used in the same order as well. All task types employed function and data in the top three most used categories. A difference was seen in the "Help Request" task type where control was the most used category. When providing help it emerges that control flow of the program is more important to explain than the actual data values. Another difference was seen in the "Proposed Changes" task type, where the meta category was the second most used category. This is intuitive as the changes are being discussed, while the reasoning behind such changes must be negotiated between the programmers, hence the meta category emerged as being heavily used.

It appears that both function and data are the most important categories no matter what the task type. Differences appear when help is being provided, in those cases it emerges that the control flow of the program is more important to understand than any other category when asking for help.

When changes are being proposed, the programmer's own reasoning earns more importance as was seen by use of the meta category. State-high also became important when explaining code, providing help or describing bugs.

All task types opened the summary with a function type statement except for the "Help Request" task type which opened with a data type statement most often.

Conclusion

The study set out to discover category usage patterns in program summaries belonging to experienced programmers. Open Source mailing lists were used and provided a wealth of information. Experienced programmers are often attracted to open source development and the programmers studied here were frequent contributors. Advantages are also gained as the programmers are not influenced by the 'presence' of an experimenter. We also examined the types of categories used depending on the task being performed. These results were successfully gathered and have implications for the design of the described SV tool, which will now be described.

The data abstraction and the higher abstractions need to be closely integrated to allow easy switching between them. The control abstraction also needs to be integrated with the data abstraction, as the control category most often interacted with the data category.

No matter what the task being performed is, both function and data abstractions will be needed. The state-high category is important for tracking down bugs along with the data and function categories. When investigating the code in order to make changes, the programmer defends their reasoning and this is seen through the heavy use of the meta category. This supports a feature such as annotated views. For example, a "post-it" type feature that allows the programmer to make notes (aside from comments) that are visible within in the integrated development environment and can be used a means of guidance when the code is being discussed between one or more additional programmers.

Acknowledgements

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

Many thanks goes to Dr. Judith Good for both guidance and helpful feedback.

References

Ball, T.J. and Eick, S.G. (1996) Software visualization in the large, *IEEE Computer* 29(4) 33-43

BSF Project, Available at: http://jakarta.apache.org/bsf

Gold, N. and Auslander, G. (1999) Newspaper coverage of people with disabilities in Canada and Israel: an international comparison, *Disability & Society*, Vol 14, No. 6, pp. 709-731.

Good, J. (1999) Programming paradigms, information types and graphical representations: empirical investigations of novice comprehension, Ph.D. Thesis, University of Edinburgh.

James Project, Available at: http://james.apache.org

jEdit Project, Available at: http://www.jedit.org

Lucene Project, Available at: http://jakarta.apache.org/lucene

Slide Project, Available at: http://jakarta.apache.org/slide