The Application of Content Analysis to Programmer Mailing Lists as a Requirements Method for a Software Visualisation Tool

Pamela O'Shea and Chris Exton
Software Visualisation and Cognition Research Group,
Department of Computer Science and Information Systems,
University of Limerick, Ireland
pamela.oshea@ul.ie

Abstract

The study set out to examine the following research question: 'What types of information are most important to the experienced programmer during maintenance ?'. A content analysis scheme was applied to program summaries extracted from online open source Java mailing lists in order to investigate the information types employed when describing programs. The aim of which was to explore a method of requirements gathering for supportive software visualisation tools. The summaries were examined collectively and also as part of their respective task type categories. It was found that informal programmer comments and data type descriptions were most important.

1. Introduction

The content analysis method has been widely employed in many disciplines including anthropology, ethnography, history, linguistics, literature, political science and psychology [11, pp. 11-12]. Krippendorff explains that contributions from such diverse areas have 'broadened the scope of the technique to embrace what may well be the essence of human behavior: talk, conversation, and mediated communication'.

Krippendorff defined content analysis as the following and it is perhaps the most widely accepted definition of content analysis:

> *Content analysis is a research technique for making replicable and valid inferences from texts (or other meaningful matter) to the contexts of their use* [11, p. 18].

In this study, inferences regarding the content of program summaries authored by experienced programmer's were made. The aim of which was to provide a complementary means of requirements gathering for supportive software visualisation tools.

Software visualisation tools are often developed based on the intuitions of the programmers who develop them, and are not necessarily based on observations of the experienced programmer. When observations are carried out, they often involve very few programmers due to the difficulty of recruitment, and therefore suffer from problems of scope. Gathering such data using experimental 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 are often created by the experimenter and can be artificial. Even in more immersive approaches where the experimenter works within the participant's workplace, the presence of an unfamiliar person changes the nature of the experience. In both cases, the environment is unnatural and the participant is aware of being monitored.

The investigation described in this paper was designed to examine the types of information present in program summaries authored by experienced programmers. The types of information present within the program summaries indicate the information types of most value to the programmer, hence implications arise for the types of information which should be supported within supportive software visualisation tools. In this way, the appropriate types of textual and graphical abstractions of the software system can be provided within tools. For example, it emerged from the study that *data* and *meta* type categories were frequently used in conjunction with each other. This implied that *data* type information should be supported within the tool with the expectation that informal programmer notes (*meta*) will be required as a followup.

In gathering data of this type, it is also possible to investigate the strength of relationships between the type of task being performed on the program and the types of informa-

Proceedings of the 2004 International Workshop on Software Technology and Engineering Practice (STEP'04)
0-7695-2293-9/04 $20.00 © 2004 IEEE

tion present in the program summary.

Using content analysis on open source mailing lists provides a wealth of information while overcoming many of these issues for our particular research investigations. The medium of email list communication, was described by Mockus et al. [12], as the primary means of communication for open source projects 'where developers work in arbitrary locations, rarely or never meet face to face, and coordinate their activity almost exclusively by means of email and bulletin boards'. Hence, the mailing list medium can be viewed as containing a substantial proportion of the information passed between developers of the project, making mailing lists a rich source of data.

Mockus et al. described two case studies of the email archives of both the Apache and Mozilla open source systems. The authors stated that good open source software (OSS) is at least on a par with the quality of industrial projects, in particular, open source software was described as setting forth 'a serious challenge ... to the commercial software businesses that dominate most software markets today'. In addition, some 'OSS development are often claimed to be equivalent, or even superior to software developed more traditionally'.

The volunteer developers possess a unique motivation where 'Code is written with more care and creativity, because developers are working only on things for which they have a real passion'. In many cases, a large number of developers contribute to any one project and a major motivational influence is that 'Work is not assigned; people undertake the work they choose to undertake'.

As already stated, the investigation described here is motivated by the need to investigate the most important types of information for supportive software maintenance tools. Software visualisation was concisely described by Ball and Eick 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 [5, p.2].

A common goal for visualisations was identified by Stasko et al. [17, p. xi] as 'transforming information into a meaningful, useful visual representation from which a human observer can gain understanding'.

Typical real-world software systems are quite complex and difficult to comprehend. An in-depth knowledge of such a system requires significant investments of time. As a result, a large cognitive load is placed on the maintainer of such code. Software visualisation aims to help the programmer carry such a cognitive burden. Consequently, software visualisation tools have much to gain from the results of the program comprehension studies.

This paper describes a study that analysed programmers' summaries found in open source Java mailing lists. A mailing list was searched using the keyword 'summary', the resulting messages were then examined and the program summaries extracted from the messages. While this method (as with any other method) cannot alone generate definitive answers and should be compared to results generated from other methods, it does provide many benefits which are described in Section 2. The motivation behind the study is discussed further in Section 3. The study itself is detailed in Section 4, while the results are presented in Section 5. Finally the conclusions and future work are both presented in Section 6.

2. Benefits of Mailing List Analysis

Software visualisation tools are often evaluated by comparing the levels of program comprehension achieved by their respective users during controlled experiments.

One could argue, however, that the 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 and unnatural environment and will quite possibly function and behave differently than under more normal circumstances, i.e. experimental controls have potential to lack ecological validity.

On the other hand, options are limited when dealing with professional programmers in an industrial setting. Immersive approaches such as action research are sometimes possible. In doing so, research can be performed within a company over an extended period of time and be used to gather data from the programmers' own environment. The long term access necessary for such a study, however, may be seen by the company in question as being overly intrusive, and disruptive with respect to their productivity. Consequently, it can be difficult to achieve access to larger numbers of programmers.

The method described within this paper can be used to complement such approaches and facilitate a comparison of results with the added benefits of increased ecological validity and access to a larger sample size.

3. Motivation

Our objective can be stated in the form of a research question as follows:

'What types of information are most important to the experienced programmer during maintenance ?'

For the purposes of this study, we are most interested in finding the context in which program items were described by the programmers in their summary accounts

(e.g. within a data flow or control flow context). These information types were recorded using the categories from the program comprehension scheme described in Section 4.1. The results will provide feedback for the most important types of information needed within supportive tools. Appropriate textual and graphical representations of the software can be used to abstract the system into many views, which can be presented to the user through the visualisation tool for further investigation.

The language under investigation for this study was Java and a number of reasons led to this decision. Firstly, Java is an object oriented language allowing for this paradigm to be studied further. Secondly, a large number of active open source Java projects exist. Finally, software visualisation tools are well supported within Java environments due to the JPDA [1] and environments such as Eclipse [6] and Netbeans [13], which allow for the integration of multiple tools into a single uniform environment.

As a result, Java mailing lists were examined during the study allowing for the examination of the programmer summary accounts. An analysis of these summary accounts allowed inferences to be made regarding the numbers and types of information employed by the programmers when working on the projects. The following section will describe this study.

4. The Study

A stream of laboratory research requires the participant to write a program summary during the experiment, which is later analysed by the researcher and/or independent analysers. Such studies were performed by Pennington [14], [15] and later developed by Good [9]. While one of Pennington's studies investigated expert programmers, Good studied novice programmers as this was the type of participant under study for that particular investigation. Both researchers investigated non-object oriented programs[2]. Pennington's work was built upon by Good and was found to be difficult to replicate due to missing information and coding manuals. Good refined the coding scheme, adding the benefits of replicability and reliability through the use of coding manuals and decision trees. The schema for analysis of program summaries developed by Good was found to be the nearest study within the literature, which resembled the study of information types in relation to the research questions of this paper. Consequently, the schema was examined for the investigation and an adapted version of Good's program comprehension scheme [9] was employed to analyse the program summaries which were collected from the mailing lists. [3]

Adaptations to the scheme were required in order to adapt it to object oriented program summaries and the mailing list domain. The adapted scheme which will be discussed shortly, is an appropriate starting point in helping us to find the information types employed by programmers during maintenance. It also facilitates measurement of the usage frequency of the information types and their usage patterns.

This adapted schema was applied to the program summaries in a content analysis manner. That is, all the guidelines for performing content analysis studies were observed. A coding manual and a set of decision trees were developed with the coding scheme and are available upon request. The decision trees can be used in cases where the coder is unsure if a segment belongs to one or more categories as the tree will guide the coder to correct category choice. In order to measure the reliability of the coding, tests exist to quantify the confidence level of the reliability of the study and should be especially important when considered within the context of experimental replication. The Kappa test was employed to measure reliability for this study and will be reported upon in the results section.

The following section describes the process of selecting appropriate mailing lists for investigation and then the selection of program summaries for analysis. The section also describes the application of the program comprehension scheme to the summaries.

4.1 Procedure

A consistent method for the collection of program summaries was employed. Open source Java mailing lists were identified[4] and summaries were chosen from four lists. Lists were selected alphabetically until the number of gathered summaries reached fifty (an additional eight summaries were added from the Kappa reliability test). The alphabetical selection meant that the projects were listed alphabetically and projects were examined in that order for program summaries. A small number (less than five summaries) of summaries were rejected as only summaries of length three lines and more were chosen for analysis since there were many longer and more comprehensive summaries to choose from.

[1] Java Platform Debugger Architecture is an API which allows tool developers to monitor the dynamic activities of the Java virtual machine, thus having the potential to support realtime visualisations.

[2] Pennington investigated COBOL and Fortran program summaries and Good investigated Prolog and Visual Language program summaries.

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

[4] Originally the Sourceforge [16] home for open source projects was employed, where selections were made from the most frequently downloaded projects, however, it was found that all of the Java projects were pointing to another location, the Apache Jakarta site. Hence, the lists were chosen from the Jakarta [10] site which is a home for open source Java projects within the open source community, run by the Apache Software Foundation.

The names of the four development mailing lists used were *Commons* (a jakarta subproject focused on all aspects of reusable Java components [1]), *Jetspeed* (an open source implementation of an enterprise information portal, using Java and XML [2]), *Log4j* (allows logging at runtime without modifying the application binary [3]), and *Oro* (a set of text-processing Java classes that provides Perl5 compatible regular expressions, etc. [4]).

Within each list, the posts were listed from the most recent to the oldest. Threads were searched for messages that contained program summaries using the keyword 'summary' as the search criteria. In the majority of cases, the summaries were contained within submitted bug reports. This identification of program summaries was performed manually as each returned post had to be read in order to verify if it described a program or a section of code in any way. Those posts which were not concerned with describing a program or a section of code were rejected.

Prior to coding, the summaries were split into segments, with each segment consisting of a subject and a predicate (either of which may be implied). For example, the following statement was split into segments as shown below (further segmentation examples are provided as part of the coding manual, which is available upon request):

'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.'

Two types of variables were gathered from the program summaries: structural variables and content variables (similar to Gold and Auslander [8]). Structural variables were variables gathered from the message itself, i.e. the mailing list name, the message subject, an assigned identification number, the reason for the summary (task type). Content variables consisted of refined categories originally from Good's program comprehension scheme plus an additional three categories (*bucket*, *code* and *locator*). These fourteen categories are termed the *refine one* categories. Three stages were involved in the content analysis process (referred to as *refine one*, *refine two* and *refine three* throughout the remainder of the paper).

Stage one of the coding process involved assigning one of the *refine one* categories to each segment. Stage two involved assigning one of *refine two* categories to the segments. Finally stage three involved assigning one of the *refine three* categories to the segments. It should be noted that each segment has a single *refine one* category assigned to it. Then, depending upon the *refine one* category assigned to a segment, the segment may or may not have a single *refine two* and a single *refine three* category assigned to it. This is due to the fact that refinements were only made to the *function*, *action*, *data*, *control*, *meta* and *locator* categories. As a result, segments categorised under the other categories will not have *refine two* or *three* categories assigned to them. The *refine one* categories will now be enumerated with an explanation and example where appropriate.

- **Function** (F): The overall aim of the program, described succinctly. This category also includes any functionality descriptions about packages, components, classes, objects, methods, variables, algorithms and files. 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. This category also includes any data descriptions about packages, components, interfaces, classes, objects, methods, variables or external data sources (e.g. streams, program output). Example: *'it remains as zero'*.

- **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. This category also includes control descriptions about threads, methods, program control (delays, external, ordering) and any java virtual machine initiated events (garbage collection, errors, exceptions). Example: *'But instead it exits'*.

- **Unclear** (U): Statements which cannot be coded because their meaning is ambiguous or uninterpretable. 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. This category also includes meta descriptions about the code itself, causes of problems and any other comments made by the programmer. 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)'*.
- **Bucket** (B): A segment that could not be classified into any of the other given categories may be classified as *bucket*. The *bucket* category exists in order to avoid the forced coding of a segment into an unsuitable category.
- **Locator** (L): More specific to mailing lists, usually occurs in the opening statement of a summary or a new paragraph. This category includes any locators that are used to point to packages, processes, components, interfaces, classes, objects, methods, variables, code excerpts, single lines of code, files or external sources. Example: *'In class org.apache.commons.dbcp.BasicDataSource'*.
- **Code** (CD): Occurs if more than a single line of code appears in succession.

Each segment was examined and categorised into one of the fourteen categories above (*refine one* stage). The next step involved applying refinements *two* and *three* if necessary. Due to space constraints all of the refinements cannot be enumerated here, however the results section clarifies the interpretation of the *refine two* and *three* categories. Since the *data* category is important in the results, a summary will now be provided of its refinements.

Refine two of the *data* category involved recording *what* data items the programmers were writing about. That is, whether the programmer was discussing, for example, a package, a class, an object, a variable. *Refine three* then involved recording *how* the programmer discussed the data item in question. That is, whether the programmer described the *data flow* (if the input/outputs of the data are described), *data state* (if the expected state of the data item is described and compared to what the state should actually be), *data structure* (if the storage or layout of the data item is described), *data type* (the type of the data item), or the *data value* (the value of the data item is discussed) of the item in question.

The results from the fifty-eight summaries are provided in the next section. Each of the fifty-eight summaries were divided into one of five task types and then analysed within their task type groups. The task types were generated from the data found within the mailing lists and include, bug descriptions, modifications, tests, system criticisms and help requests. For example, if a summary is classified as a bug description then the overall aim of the summary was to describe a bug.

5. Results

The first author and an independent coder analysed eight of the fifty-eight summaries in order to measure agreement after many coding and segmentation conventions were documented. A Kappa of 0.8818 was recorded. The first author carried out the analysis of the remaining fifty summaries using these agreed conventions. According to the El Emam threshold table [7], this Kappa result equates to 'excellent'.

It should also be noted that the author and the independent coder initially agreed 89% of the time on segmentation. However, after the assumptions were documented and adhered to, both coders agreed in all cases. The first author also used these segmentation assumptions for the remainder of the study.

The following sections present and discuss the results of the types of information discussed within the fifty-eight program summaries, as well as the types of information discussed for each task type. Finally, any repeated patterns which were recorded will be reported upon.

5.1 Fifty-Eight Summaries

5.1.1 Information Types

The most frequent information types (*refine one* categories) described within the 58 summaries are shown in Figure 1. The results show that the three most frequently employed categories were, *meta* at 24.4%, *data* at 23.5% and *control* at 10.3%. It was also worth noting that the *locator* category was frequently employed at 8.7%.

The recorded *refine two* categories for each of the information types showed that the most frequently used were, *meta code* at 23.7%, *data variable* at 8.4% and *data class* at 7.1%.

Finally, the recorded *refine three* categories showed that the three most important categories were, *meta code behaviour* at 12.1%, *meta code modification* at 10.9% and *data class structure* at 9.7%. It is also worth noting that the next two most frequently used categories were *data variable state* at 5.8% and *data method flow* at 4.1%.

From these results it was seen that *meta*, *data* and *control* type descriptions were important for the 58 summaries. The *meta* categories were often describing the behaviour of the code, while the *data* descriptions were often discussing

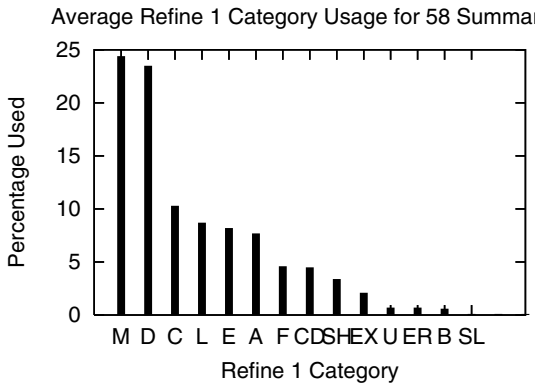

Figure 1. Frequency of Category Usage for 58 Summaries

the structure of the classes, followed in popularity by descriptions of the state of variables and the data flow in and out of the methods.

Meta Category

The high levels of *meta* indicate the importance of informal comments for program summaries within mailing lists. Informality and the ability for the programmer to create their own custom notes is mostly an unaddressed issue within the tool design guidelines of the literature. Documentation within existing systems is on a formal level and comments usually become a longterm fixture rendering them unsuitable for short term informal notes or even task specific notes. External documentations or files bring with them the effect of occlusion, again making them unsuitable for notes which need to be attached to specific areas of the source code or even diagrams.

The Rigi tool touched upon this issue in a minor way, for example Systa et al. discussed the notion that the

> static dependency graphs of a subject system can be annotated with attributes, such as software quality measures, and then be analyzed and visualized using scripts through the end-user programmable interface [19].

This feature within the Rigi tool, however, is not a common feature within other software visualisation tools. Also, it does not support the extent to which informal notes were found during the program summaries.

Informality, in the form of user annotations was also recommended as part of Storey et al.'s fourth design element [18], where layers of annotations were discussed as a feature for support of hypothesis driven comprehension. In particular, a call was made to support the creation of a chain of hypothesis and resulting sub-hypothesis, as well as the ability to record postponed or discarded hypotheses. While this was published as a design element, it has not been widely adopted by tool designers to date.

Informal comments need to be addressed in many ways, for example they can be temporary short term notes or more long term notes, task specific or general, shared with peer developers for feedback or for private use. Either way, their popularity within the online program summary accounts was noteworthy. These mailing lists are effective as the primary means of communication among peer developers of open source projects, hence, the importance placed on informal programmer comments and notes should be taken into account.

Data Category

The most frequent types of data descriptions showed that the structure of the classes were important, as well as the changing state of variables. Additionally, the data flow to and from methods was important.

It was interesting to note that it was the class level which was discussed most often within the data context. Thus indicating the level of navigation employed by the programmers. In particular, the structure of the class was most important, this included information regarding the internal structure of the classes. For example the location of constructors, methods, variables. The structure of the class definition itself was also recorded by this result, that is, whether the class was declared, for example as public, private or protected. Many integrated development environments, such as IBM's Eclipse and Sun's Netbeans provide such summary information for both methods and classes in the form of a tree listing. This result shows the importance of such features and perhaps the need to allow the user to customise their own collection of frequently referenced or important classes during maintenance.

Following the classes, it was variables which were discussed frequently. This result demonstrates that the changing state of the variable over time was more important than any other information regarding the variable, for example, the internal structure or type of the variable. The state of the variable includes comparisons to older values, descriptions of scope and or lack of initialisation. Hence, the history of a variable's previous values should be easily accessible to support this.

Within the context of data descriptions, it was also found that data flow to and from methods was important. This result shows the importance of being able to monitor the flow of parameters in and out of methods and the return variables. This information was more important than the layout of the method itself.

Control Category
The *control* type statements were used to discuss method invocations and the order of execution within the program itself. Descriptions of java virtual machine (jvm) initiated exceptions were also frequent.

These results show the importance of method control flow information, which refers to the calling of methods, the order in which they were called and the location where they were called from. This highlights the importance of control flow information regarding methods when describing problems, as well as the ordering of events within the program.

Following these, Java virtual machine exceptions were frequently described. That is, the control descriptions were interrupted by thrown exceptions from the Java virtual machine. This result highlights the importance of monitoring exceptions from the Java virtual machine during maintenance while maintaining context.

Locator Category
As noted previously, the *locator* statements were also frequent and were used to point to classes, methods and code excerpts.

The specific locators to classes combined with the previous discussion of the data structure of classes being important, this result shows the importance of classes as navigation points when performing maintenance. Thus, access to specific locations within classes must be easily accessible and identifiable.

Following the identification of specific classes, pointers to methods and code excerpts were recorded. While these were not as frequent as pointers to classes, it shows the importance of navigation and identification of methods and code excerpts selected by the programmers during maintenance and the need to support such navigation.

5.1.2 Patterns

Patterns of category usage were examined within the summaries. A pattern was defined as being contained within a program summary and could not be the last segment of one summary followed by the first segment of a new summary. No restriction was placed on the length of a pattern, that is, they were not limited to just two consecutive recurring segments but could be any number. For example three consecutive descriptions of control could be found to be a common pattern and would be denoted by the code 'control control control'. The consecutive categories did not have to be the same category types but could differ, for example, 'control-data', indicated control descriptions were often followed by data descriptions.

The three most frequently recorded patterns for the fifty-eight summaries were *meta-meta* at 9.2%, *data-data* at 8.9% and *meta-data* at 4.3%.

The patterns found shows that the programmers often followed *meta* statements with other *meta* statements. Since *meta* statements comprise of programmer comments and opinions, support again can be seen for multiple annotated type views. Such views act like graphical *post-its* where the programmer can record and attach personalised explanations (less formal than comments). Annotated views can also be a means of communication between developers as they provide a summary of the programmers understanding of the system. The *data* statements were often followed by other *data* statements which shows the need to facilitate consecutive data queries.

The *data* descriptions were used to describe variable states, class structure and data flow in/out of the methods. The *control* descriptions were used to describe method invocations, exceptions and delays within the program.

It is interesting to note that *control* statements were often followed by other *control* statements. Again supporting the need for multiple control type queries.

5.2 Task Types

The fifty-eight summaries were categorised according to their task type (purpose of the summary) and will now be presented as such. The purpose of the summaries was examined by the author and each summary was found to either described a bug, a modification, a test, a criticism or to ask for help. The three most frequently cited categories are given and in some cases interesting categories which are also frequently employed are listed.

The summaries were distributed as follows: 35 bug description task types, 10 modification task types, 4 testing task types, 4 system criticisms task types, 5 help requests.

Table 1 contains the pattern results for each of the tasks, while Tables 2, 3 and 4 contain the category results from the three coding stages.

Task Type	Pattern Usage Frequency
Bug Descriptions	meta meta:9.7%, data data:8%, control control:5.6%
Modifications	data data:12%, meta data:7.3%, data meta:5.3%
Testing	meta meta:11.4%, data data:5.7%, elaborate data:4.5%
System Criticisms	data data:12.8%, data state-high:6.4%, state-high meta:6.4%
Help Requests	meta meta:18.6%, data data:9.3%, locator meta:7%

Table 1. Pattern Usage for Task Types

Task Type	Refine 1 Usage Frequency
Bug Descriptions	meta:23%, data:22.6%, control:15.7%
Modifications	data:28.1%, meta:19.4%, elaborate:11.9%
Testing	meta:23.9%, data:20.7%, locator:15.2%
System Criticisms	data:27.5%, meta:23.5%, action:11.8%, state-high:11.8% control:9.8%
Help Requests	meta:40.7%, data:20.9%, locator:12%

Table 2. Refine 1 Category Usage for Task Types

When describing bugs, the *data* descriptions were used to describe variable states, class structure and data flow in and out of the methods. The *control* descriptions were used to describe method invocations, exceptions and delays within the program.

For the summaries describing modifications, the *data* descriptions were used to describe classes, variables and objects. As with the bug descriptions, the structure of classes were again important. In addition, the values of variables and the structure of objects were also frequently described.

During the summaries describing tests, the *meta* descriptions were used to describe the modifications of the code followed in popularity by the code behaviour and comments about the tests. The *data* descriptions were again used to describe the structure of classes followed in popularity by the state of the variables, as well as program output. The *locator* statements were used to point to both classes and java source files during discussions of tests.

The *data* descriptions for system criticisms were again used to describe the structure of the classes and the data flow to and from the methods.

When requesting help, as with the descriptions of tests, the *meta* descriptions were used to describe the behaviour of the code followed by the modifications to the code, as well as the programmers' own reasoning. It was interesting to note that the programmers' own reasoning was present within the help requests, this was to be expected as the programmer was explaining where they had problems.

As with the bug descriptions and modifications, the *data* descriptions for the help requests were used to describe the variable values. Again, similar to the modification accounts, this was followed in popularity by the structure of objects.

In summary, the *meta* and *data* categories appeared in the top three categories used for all the task types. However, differences between the task types can be seen in the third most frequent category. The *control* category was the third most popular category in the bug descriptions, while the *elaborate* category was the third most popular category in the modifications. Both the testing summaries and help requests had *locator* statements being the third most popular categories used. In contrast the summaries which were critical of the code had *state-high* categories being the third most frequently used. Such common traits and differences have potential to guide the requirements for supportive maintenance tools, where appropriate views can be preset or recommended to the programmer depending upon the task type at hand.

Task Type	Refine 2 Usage Frequency
Bug Descriptions	meta code:20.5%, data variable:9.8%, control method:8.5%, action:10%, function:9.4%, locator:5.6%
Modifications	meta code:23.3%, data class:13.6%, data variable:8.7%, data object:6.8%
Testing	meta code:26.9%, locator class:7.5%, locator file java:7.5%, function class:7.5%, data class:7.5%
System Criticisms	meta code:27.8%, data class:19.4%, data method:11.1%
Help Requests	meta code:33.8%, meta user:9.9%, data variable:7%, meta cause-of-problem:7%

Table 3. Refine 2 Category Usage for Task Types

6. Conclusion and Future Work

Future work includes increasing the sample size in order to gather a larger number of summaries in other task types which are not bug descriptions thus allowing stronger generalisations to be made. Future work also has the potential to examine differences not only between task types but also between individual programmers by collecting data posted by the same authors. Also, by documenting many assumptions regarding the *meta* category, an improved kappa of 0.9449 was found to be possible. Future work includes aiming towards this level of coder agreement with at least three independent coders.

The results from studies employing this analysis method

Task Type	Refine 3 Usage Frequency
Bug Descriptions	meta code behavior:11.7%, data variable state:8.5%, meta code modification:7%, data class structure:6.6%, control jvm exception:5.2%
Modifications	data class structure:17.7%, meta code behavior:10.1%, meta code modification:7.6%, data external flow:6.3%, data object structure:6.3%
Testing	data code modifications:23.3%, data class structure:11.6%, meta code behaviour:9.3%, meta code testing:9.3%,
System Criticisms	data class structure:21.4%, meta class modidication:17.9%, data method flow:10.7%,
Help Requests	meta code behavior:21.6%, meta class modidication:17.6%, data variable value:9.8%, data object structure:7.8%

Table 4. Refine 3 Category Usage for Task Types

can be used to complement the studies from the literature, in order to guide software maintenance tool designers. It has emerged from the findings that *data* type descriptions are vital to the programmers understanding of the system. This may not be surprising but significantly the type and nature of the data queries have emerged from the study results. The most important *data* descriptions were those which described the structure of classes (for task types modifications, testing and code criticisms). Variable states were important for the bug descriptions and the testing summaries while the variable values were important for the help requests.

Meta descriptions were also frequently used in conjunction with data. These results indicate that *data* type representations of the system should be implemented and integrated closely with views designed to represent the *meta* category, e.g. annotated views.

In conclusion, an adapted content analysis method has been applied to real data from online mailing lists which facilitates access to a greater sample size, which would otherwise not have been possible through laboratory studies alone. Other benefits include the examination of data from experienced programmers within their natural environment. A wide variety of program summaries may be extracted from varying task types, programmers and projects which facilitates a rich source of requirements for the design of supportive tools.

7. 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. Jim Buckley for informative discussions and helpful feedback. Also a special thank-you to Dr. Judith Good for dedicating time and providing much valued feedback during the study.

References

[1] Commons project. *Available at: http://jakarta.apache.org/commons/*.

[2] Jetspeed project. *Available at: http://portals.apache.org/jetspeed-1/*.

[3] Log4j project. *Available at: http://logging.apache.org/log4j/docs/*.

[4] Oro project. *Available at: http://jakarta.apache.org/oro/*.

[5] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[6] Eclipse. Eclipse Integrated Development Environment. 2003.

[7] K. E. Emam. Benchmarking kappa for software process assessment reliability studies. *International Software Engineering Research Network Technical Report*, 98(02), 1998.

[8] N. Gold and G. Auslander. Newspaper coverage of people with disabilities in canada and israel: an international comparison. *Disability and Society*, 14(6):709–731, 1999.

[9] J. Good. Programming paradigms, information types and graphical representations: Empirical investigations of novice comprehension. In *Ph.D. Thesis*. University of Edinburgh, 1999.

[10] Jakarta. Jakarta Apache project: directory of open source java projects. *Available: http://jakarta.apache.org*, 2003.

[11] K. Krippendorff. Content analysis: An introduction to its methodology. *Sage Publications*, Second Edition, ISBN: 0-7619-1545-1, 2004.

[12] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[13] Netbeans. Netbeans Integrated Development Environment. 2003.

[14] N. Pennington. Comprehension strategies in programming. In *G. M. Olson, S. Sheppard, and E. Soloway, editors, Empirical Studies of Programmers: Second Workshop (ESP2)*, pages 100–113. New Jersey. Ablex Publishing Corporation, 1987.

[15] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[16] Sourceforge. Sourceforge project: directory of open source projects. *Available: http://sourceforge.net*, 2003.

[17] J. Stasko, J. Domingue, M. Brown, and B. Price. Software visualization: Programming as a multimedia experience. MIT Press, 1998.

[18] M.-A. D. Storey, F. D. Fracchia, and H. A. Muller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th IEEE International Workshop on Program Comprehension (IWPC'97)*, page 17, Dearborn, MI, USA, May 28-30 1997. IEEE.

[19] T. Systa, P. Yu, and H. Muller. Analyzing java software by combining metrics and program visualization. In *Conference on Software Maintenance and Reengineering*, pages 199–208. IEEE, Feb-Mar 2000.