Program Comprehension: Investigating the Effects of Naming Style and Documentation

Scott Blinman **Andy Cockburn**
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
{sdb47, andy}@cosc.canterbury.ac.nz

Abstract

In both commercial and academic environments, software development frameworks are an important tool in the construction of industrial strength software solutions. Despite the role they play in present day software development, little research has gone into understanding which aspects of their design, influence the way software developers use frameworks at the source code level.

This paper investigates how the comprehensibility of an application's source code is affected by two factors: the naming styles for framework interfaces, and the availability of interface documentation. Results show that using a descriptive interface naming style is an effective way to aid a developer's comprehension. Documentation also plays an important role, but it increases the amount of time a developer will spend studying the source code.

Keywords: Software development frameworks, program comprehension, naming style, documentation.

1 Introduction

A Software development framework is an integrated set of reusable software components designed for a specific application domain. These components represent a semi-defined application, which are customised by an application developer (user) in-order to build a complete application (Brugali 2000).

Development frameworks are often accompanied by an Integrated Development Environment (IDE), which provides the user with a graphical interface for the framework's functionality. An IDE will generally be composed of a range of development tools designed to increase the user's productivity. Such tools may include: a documentation viewer, debugging tool, visual programming environment and interface design tools. In principal, these tools aim to simplify the task of developing software applications, by reducing the amount of source code the user will need to type. However, despite efforts by framework vendors to provide users with efficient tools for developing applications, the user will at some point, be required to inhabit the framework at a source code level. This level of interaction usually becomes necessary as the complexity of the application being developed increases.

Programming source code requires the user to have a more in-depth understanding of the fundamental design of the framework. The amount of effort the user will need to invest into developing this understanding will ultimately depend on how well the framework is designed. This raises some interesting questions regarding which aspects of a framework's design affect how intuitive it is to use at the source code level.

Extensive prior research has been done in developing the concept of frameworks, and formalising ways to engineer them. A well designed development framework should be constructed to closely encapsulate the problem domain which they are intended to solve (Roberts 1997, Schmidt 2003), and robust enough to be used in industry for the development of large software applications. There are many papers which evaluate how development frameworks interact with operating systems, and machine hardware. These comparisons may include how efficient certain framework components are at retrieving information from a database (Microsoft 2001), or how effective one may be at rendering 3D graphics. However, there has been no research into which aspects of a framework's design, effect how well users comprehend the source code of an application which have been developed with the aid of a framework.

The objective of the research presented in the paper is to gain a better understanding of the factors which affect a user's interaction with a framework at the source code level. This research investigates how the comprehensibility of source code is influenced by the naming style used for a framework's interfaces, and the availability of interface documentation. Gaining a better understanding of these aspects will help framework designers build development environments that are more productive, and help application developers build software which is easily maintained.

2 Related Work

Prior research into program comprehension has provided a basis for this study. Two areas which are particularly relevant include research into theoretical models of program comprehension, and the development of methods for measuring program comprehension.

2.1 Theories of Program Comprehension

Over the past thirty years, many studies have been conducted into the human psychology of computer programming. In particular, this work includes understanding how programmers comprehend source code from a cognitive and perceptual perspective. From this research, two main theories of program comprehension have emerged.

The earliest model of program comprehension, commonly known as the ‘bottom-up’ approach, was proposed by Shneidermann (1977) and further developed by Shneidermann and Mayer (1979). This theory essentially describes program comprehension as a data driven process. The programmer uses their syntactic knowledge of a particular programming language, together with semantic knowledge (understanding of general programming techniques and knowledge gained through experience), to assign meaning to small fragments of source code. Each small piece of knowledge is easily remembered, and the programmers understanding of the whole program is increased as they comprehend more parts of the program. Eventually, through this iterative process, a large part or the whole program can be confidently understood. Shneidermann proposed that descriptive or mnemonic variable names help the programmer to comprehend the program syntax more easily, by placing less of a "burden on the programmer to encode the meaning of the variable". Descriptive variable names help reduce the program’s complexity, greatly simplifying the programmer’s comprehension overhead.

A later and more popular model of program comprehension commonly known as the ‘top-down’ approach was first proposed by Brooks (1983). This model is a more conceptually driven concept, where a programmer starts with a general hypothesis about what the program does. The formation of this initial hypothesis is often facilitated by sources outside the program, such as high level documentation providing a description of the program’s function. The initial hypothesis will give the programmer some idea of what to look for when they begin to study the source code. Brooks believes that programmers do not necessarily read a program line by line, but instead scan the source code searching for 'beacons', which they use to elaborate their current hypothesis by forming more specific, sub-hypotheses. Over time the programmer will develop a hierarchical structure of hypotheses, beginning with the initial hypothesis at the top, followed by more refined subsidiary hypotheses which are more closely bound to specific parts of the programs source code. The larger the hypotheses hierarchy a programmer develops, the more the programmer understands the function of the application the source code represents.

An important link between the ‘bottom-up’ and ‘top-down’ models is the presence of beacons within the source code. A beacon is defined as "sets of features that typically indicate the occurrence of certain structures or operations within the code" (Brooks 1983). This broad definition for one of the more important aspects of the ‘top-down’ model, resulted in widespread research into identifying what features of source code serve as beacons. Wiedenbeck (1986) empirically verified that recognizable patterns within the source code, which serve as indicators of a stereotypical structures or operations, can be considered beacons. Further work by Gellenbeck and Cook (1991) showed that the definition of a beacon can also be extended to include descriptive procedure names and variables.

Teasley (1994) showed that the presence of good naming style in a program’s source code was relied upon more by novice programmer then an experienced programmer. Novice programmers are less able to recognise beacons associated with code patterns because they lack knowledge gained by experience (semantic), and therefore rely more on naming style to bring meaning to code segments. On the other hand, experienced programmers tended to rely more on code pattern beacons rather then naming style when comprehending a program’s source code. The fact that experienced programmers have a broader experience base to draw upon enables them to use a wider range of strategies when comprehending source code. Similar findings were reported by Crosby (2002), who found that in the absence of a good naming style, experienced programmers were able to extract more information from a portion of source code, compared to novice programmers. Crosby suggested that the use of a good naming style and the inclusion of documentation such as inline comments, could speed up the comprehension time for both novice and experienced programmers.

The role that documentation has to play in assisting a programmer to comprehend source code is important. Documentation provides the programmer with a set of indicators which will help them during hypothesis verification (Brooks 1983). Indicators which could be considered documentation include: source code header information, inline source code comments, application user manuals and software library manuals (i.e. API reference texts).

Based on prior research, it is likely that a descriptive naming style for a framework’s interfaces, accompanied by interface documentation, will improve the comprehensibility of an applications source code.

2.2 Measuring Program Comprehension

Early work by Shneidermann (1977), demonstrated that program comprehension can be measured through the use of a 'memorization-recall' test. This involves giving the participant a set amount of time to study a segment of source code. After this time, the source code segment is removed from the participant’s sight. They are then asked to write out as much as they can remember about the source code segment they just studied. A measure of comprehension is then given by using an appropriate scoring system, where the functional and literal accuracy of the recalled source code are both considered. The rational behind this approach directly relates the comprehensibility of a segment of source code to how easy the source code is to memorise, and consequently recall. Brooks (1980) later published a review of current methodologies in studying programmer behaviour and associated problems. He proposed a version of the ‘memorization-recall’ test, where high-level concepts about the code segment’s global structure and algorithms, should also be considered as a measure of program comprehension. Brooks notes however, the use of ‘memorization-recall’ tests is only applicable to isolated code segments and small programs. They would serve little benefit when studying the comprehension of larger bodies of source.

3 Experimental Method

The study used a 2x2 within-participants factorial design. The factors were: naming style and documentation. Naming style had two levels of measurement: full-descriptive naming style, and non-descriptive naming style. Documentation had two levels of measurement: documentation provided, and no documentation provided. Dependant variables included measurement of comprehensibility and task completion time. A Latin Square method was used to counter-balance order of exposure to conditions.

3.1 Application Design

The Microsoft .Net development framework was used as the target framework. This framework uses descriptive names and is accompanied by good documentation. Using an existing framework eliminated the need to develop a pseudo framework and consequently reduced the experiment preparation time.

Four small (10-20 lines) win32 console applications were developed using components from the .NET framework. Each application performed a simple task involving the processing of input data taken from either standard input or the operating system (e.g. current date) and sending a result to standard output. No two applications performed the same task and each application used a different set of .Net framework interfaces.

All source code was written using the J# language syntax which is identical to the Java language syntax. This language was chosen to ensure that a majority of the participants would be familiar with the source code syntax. Before the experiment was conducted, it was known that the participants would all be postgraduate students from the University of Canterbury. It was therefore anticipated that the majority of these students would be familiar with the Java syntax, as it is taught by the university as part of their computer science undergraduate program.

After the four applications had been developed, two applications were randomly chosen to have accompanying documentation for each interface used in the application. This was taken directly from the documentation provided by the Microsoft .Net framework. One of the documented applications was then randomly chosen along with another application without documentation. These two applications were used as the test cases for tasks involving full-descriptive interface names. The interface names which appeared in the remaining two applications were then changed to non-descriptive names. The method of stripping names from the non-descriptive cases, involved renaming the interfaces in one application to 'Function1', 'Function2', 'Function3' ... 'FunctionN', (where N represents the number of the last interface). The interface names in the second application were then renamed to 'FunctionA', 'FunctionB', 'FunctionC' ... 'FunctionZ'. This ensured interfaces found in one application, did not have the same non-descriptive name as interfaces found in the other application.

The source code and documentation for each of the tasks were printed out separately on plain paper.

3.2 Comprehension Design

A modified version of the ‘memorization-recall’ test was used as the comprehension test. This test involved asking the participant to study the source code of an application (including the documentation in some cases), until they felt confident that they understood its function. There was no time limit imposed on the participant’s study time, instead the total time taken by the participant to study the source code was recorded. The source code was then removed from the participant’s sight, and a list of three high-level descriptions, each describing possible functions for the application was given to the participant. The participant was asked to read through these descriptions, before choosing one which they felt best described the function of the application that they just studied. The participant was encouraged not to guess the function of the application. If they did not know which description to choose, they were asked to indicate that they were unsure of the answer.

The comprehensibility of the source code was measured by considering how long the participant took to study the source code, and whether they chose the correct application description from the list provided

3.3 Participants

The participants used, were eleven postgraduate students from the University of Canterbury. The average age of the participants was 22. They had all been studying computer science at the time of the experiment, for an average of 4 years. 18% of the participants had previous experience as professional software developers for an average of 5 years. 91% of the participants had used Java to develop software in the past and 18% of participants had previously used the .Net development framework.

3.4 Procedure

Each participant completed a series of four source code comprehension tasks using the application source code and comprehension method described above. For each task, the experimenter presented the participant with a piece of paper containing a printed copy of the application source code. In two out of the four cases, the source code was accompanied by a second sheet of paper containing the interface documentation. The participant was asked to study the source code until they felt that they understood the function of the application. Once the participant had finished studying the source code, the experimenter removed the sheets of paper, and presented the participant with one multiple choice comprehension question. The participants were asked to answer this question, by writing their choice on the paper provided before proceeding to the next comprehension task. This process was continued until all four comprehension task were completed. The time taken for the participants to study each source code example was recorded by the experimenter.

4 Results

Multi-factor analysis of variance was performed on study times and multiple choice answers, based on data collected from a total of 44 comprehension tasks.

4.1 Study Times

The graph in Figure 1 shows the mean study times for each experimental task. Multi-factor analysis of variance on the data gathered for task study times, showed that the mean task study time for descriptive and non-descriptive interface naming styles, 81.9 (σ 53.4) and 177.2 (σ 117.7) seconds respectively, was found to be significantly different, F(1,10)=17.5, $p < 0.01$. The mean study time for tasks containing interface documentation and tasks not containing interface documentation, 179.5 (σ 107.6) and 79.7 (σ 68.3) seconds respectively, was also found to be significantly different, F(1,10)=85.3, $p < 0.001$.

A significant interaction was found between the study times for interface naming style and the availability of interface documentation, F(1,10) = 17.1, $p < 0.01$. This between factors interaction is shown in Figure 1. The interaction is interesting, as it suggests that when source-code is obfuscated through non-descriptive names, the use of documentation increases dramatically.

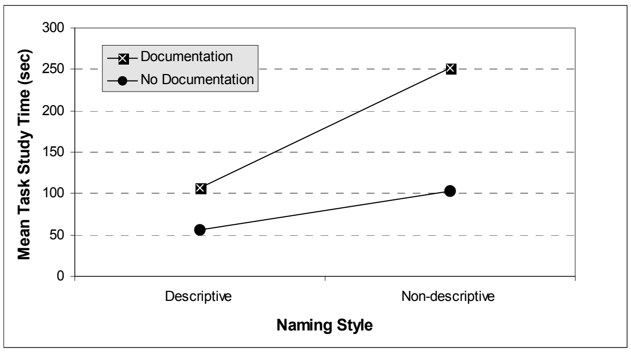

Figure 1. Mean study time between factors.

4.2 Multiple Choice Answers

The graph in Figure 2 shows the percentage of correct answers for each experimental task. Multi-factor analysis of variance on the data gathered for answers given to the multiple choice questions, showed that the mean percentage of correct answers for tasks containing descriptive or non-descriptive interface naming styles, 86.4 (σ 35.1) and 36.4 (σ 49.2) percent respectively, was found to be significantly different, F(1,10)=27.5, $p<0.001$. The mean percentage of correct answers for tasks containing interface documentation and for tasks not containing interface documentation, 77.27 (σ 42.9) and 45.45 (σ 60.0) percent respectively, was also found to be significantly different, F(1,10)=17.5, $p<0.01$.

A marginally significant interaction was found between the percentage of correct answers for the interface naming style and the availability of interface documentation, F(1,10)=4.8, p=0.053. This interaction is revealed in Figure 2, which shows that performance degraded more rapidly between descriptive and non-descriptive names in the absence of documentation.

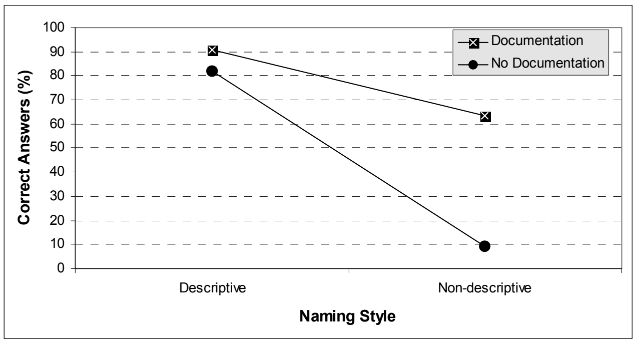

Figure 2. Percentage of correct answers between factors.

4.3 Observations

In general, the participants enjoyed doing the tasks. Their conduct during the experiment was very professional and every participant read the source code of each task thoroughly before indicating that they had finished.

When studying those tasks which included documentation, participants generally used the documentation very carefully to cross reference each interface name. However, a few participants ignored the documentation for the task involving descriptive interface names, with one participant commenting that they "never read the documentation anyway". Despite this, all participants used the documentation when studying the code for the task involving non-descriptive interface names. Some participants were confused with the meanings of various terms used in the documentation. These terms included: 'immutable', 'GetLowerBound' and 'regular expression'.

The task involving non-descriptive interface names with documentation produced the most comments from participants. Some of these comments included:

"This sucks!", "I'm getting lost." and "These are filthy names."

When participants were presented with the source code for the task testing non-descriptive interface names without documentation, most decided that the code was too difficult to understand, and quickly indicated they had finished studying. However, some participants made a more serious attempted to decipher the source code. These participants concentrated on the task more intensely than most other tasks. With one participant commenting that he would "have to memorise the structure for this one".

4.4 Discussion

For tasks testing descriptive naming style, participants were able to determine the function of an application's source code more easily then tasks testing non-descriptive naming style.

The availability of interface documentation helped participants to understand the applications source code, but increased the amount of time they would spend studying. In particular, the task which used a non-descriptive naming style with documentation, participants relied entirely on the documentation to determine the function of the source code. Consequently, this task produced the highest study times. For the task using a descriptive naming style with documentation, participants relied less upon the documentation, but still spent more time studying the source code than tasks that were not accompanied by documentation. These results suggest that the availability of documentation increases the amount of time needed to comprehend a piece of source code, rather than decrease it, as suggested by Crosby (2002).

In the absence of interface documentation, participants were able to determine the function source code using a descriptive naming style more easily then a non-descriptive naming style. However, by memorising the source code structure, rather than relying on the interface names, some participants were able to determine the function of the source code using non-descriptive interface names. This correlates with some of the conclusions found by Teasley (1994) and Crosby (2002), who found that expert programmers have more adaptable comprehension techniques, enabling them to utilise a wider range of beacons while comprehending source code.

Some concerns should be noted for the unacceptably high standard deviation calculated for some means. In particular, the task testing non-descriptive naming style without documentation had a standard error equal to the mean percentage for that task. This poor distribution of data may be due to the low number of participants in the experiment (eleven in total). Further concerns regarding the multiple choice questionnaire which was used for each task should also be noted. The fact that there was only one multiple choice question per task meant participants were either 100% or 0% correct. A questionnaire which contained more then one comprehension question would be preferable. This would produce a wider distribution of percentages, and consequently a more accurate measure of comprehension.

5 Conclusion

In the past, research into program comprehension has studied programming languages such as FORTRAN, COBOL and Pascal. The source code used has represented small fragments of an application such as a single function or a small group of sub-routines. Very little work has been done on program comprehension for an entire application, and no comprehension studies have been conducted on source code written with a software development framework.

For applications which have been written using a framework, the majority of functional, data driven logic, is encapsulated within the frameworks libraries. For example, a user of a framework would not be concerned with developing functions that perform an array sort or binary search. Instead, they will develop applications which use the array sort and binary search interfaces provided by the framework. This type of source code will therefore mainly consist of 'glue' logic, binding individual framework interfaces together to form an entire application. In this type of programming environment, many types of structural code patterns that have been recognised as beacons in the past, are likely to be absent or less pronounced. However, beacons which are common in source code that has been written with or without the use of a framework include descriptive procedure (interface) names, and forms of available documentation.

This paper described a study that investigated how the use of interface naming style and the availability of interface documentation, influences a developer's ability to comprehend source code which has been written using a software development framework.

A modified version of the 'memorization-recall' test was used to measure a user's comprehension of a series of source code examples which used either descriptive or non-descriptive framework interface names. The availability of interface documentation was also included as a component of these comprehension tests.

Results showed that developers were able to reliably comprehend the function of an application which was written with the aid of a framework that used a descriptive interface naming style and was accompanied by interface documentation. Applications written using a descriptive naming style, and not accompanied by documentation, were understood the second most reliably. Applications which were written using a non-descriptive interface names and accompanied by interface documentation, were understood the third most reliably. Application's written using a non-descriptive naming style and not accompanied by documentation, were found to be the hardest for developers to comprehend.

The amount of time a developer spends studying the source code of each application, is dependant on the availability of interface documentation. Developers spent more time studying the source code of applications that are accompanied by interface documentation, regardless of whether a descriptive or non-descriptive interface naming style is used. It was also very unlikely that a developer would correctly comprehend the function of source code which used a non-descriptive naming style and was not accompanied by interface documentation.

Although the results presented by this paper would seem obvious to most readers. The implications of this type of research bring to light the importance of naming style and documentation to both users, and designers of software development frameworks.

Modern software developers should be aware that in recent years, the definition of 'user', has adopted a broader context. The 'user' of an application includes both the people who will operate the compiled application, as well as the software developers who write, and later maintain the applications source code. Both user groups warrant careful consideration during the application design and development process.

Acknowledgments

Our thanks to the participants who helped with this study. Without their keen ability to comprehend the incomprehensible, none of this would have been possible.

References

Brooks, R. (1980). Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. Communications of the ACM, Vol 23, Issue 4, pp207-213

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies. Vol 18, pp543-554.

Brugali, D. and Sycara, K. (2000). Frameworks and Pattern Languages: An intriguing relationship. ACM Computing Surveys (CSUR), Volume 32, Issue 1es (March), Article No. 2.

Crosby, M, E. Scholtz, J. Wiedenbeck, S. (2002). The Roles Beacons Play in Comprehension for Novice and Expert Programmers. Proceedings of the Fourteenth Annual Workshop of the Psychology of programming Interest Group London. UK. pp 58-73.

Gellenbeck, E, M. (1991). An Investigation of Procedure and Variable Name as Beacons during Program Comprehension. In J. Koenemann-Bellinveau, T. G. Mohen and S, P. Robertson, Eds. Empirical Studies of Programmers: Fourth Workshop, pp 65-81. Norwood, NJ: Ablex.

Microsoft. (2001). Nile .NET vs. J2EE Benchmark. http://www.gotdotnet.com/team/compare/nileperf.aspx

Roberts, D. and Johnson, R. (1997). Evolving Frameworks: A pattern language for developing object-oriented frameworks. D. Riehle, F.Buschmann and R.C.Martin Eds, Pattern Languages of Program Design. Vol 3. Reading Mass, Addison Weley.

Schmidt, D. and Buschmann, F. (2003). Patterns, Frameworks, and Middleware: Their synergistic relationship. Proceedings of the 25th international conference on Software engineering. pp694-704

Teasley, B, E. (1994) The effects of naming style and expertise on program comprehension. International Journal of Human-Computer Studies. Vol 40, pp757-770.

Shneiderman, B. (1977). Measuring computer program quality and comprehension. International Journal of Man-Machine Studies. Vol 9, pp465-478.

Shneiderman, B. and Mayer, R. (1979). Syntactic/semanticinteraction in programmer behavior: a model and experimental results. International Journal of Computer and Information Sciences. Vol 8, pp219-283.

Wiedenbeck, S. (1986). Beacons in computer program comprehension. International Journal of Man-Machine Studies. Vol 25, pp 697-709.