UNIVERSITY OF OSLO
Department of Informatics

Effects of Expertise and Strategies on Program Comprehension in Maintenance of Object-Oriented Systems: A Controlled Experiment with Professional Developers

Master thesis
60 credits

Kaja Kværn
kajk@ifi.uio.no

May 1st 2006

Abstract

The research presented in this thesis identifies the comprehension strategies used by professional programmers with different expertise during maintenance tasks on a previously unknown application in Java. Furthermore the effects of the strategies on performance and the effects of expertise on strategies and performance are investigated. The skill to comprehend a program is useful, especially in relation to software maintenance. In order to improve software maintenance, knowledge of how professional programmers comprehend a program is important. This knowledge can be used to develop techniques to support the process, or to improve education.

We conducted a controlled experiment with 24 professional developers from five companies. The participants were introduced to a 3600 LOC Java application on which to perform three maintenance tasks. Their actions were logged and we have developed a data analysis tool to get a detailed picture of the participants' actions. Our further development of GRUMPS made it possible to see the participants' use of documentation, classes visited, when they compiled and how they executed the application. The data analysis tool can be reused by researchers in similar studies and contribute to improve the understanding of developers' program comprehension strategies. In addition to the GRUMPS data, feedback was collected through the feedback-collection method.

Two different approaches to the task solving were identified. In the first one, called *systematic* strategy, the participants tried to get an overview of the application before performing the maintenance tasks, by reading the system documentation or running the application. 75% of the participants applied this strategy. In the second one, called *as-needed* strategy, the participants went straight to the task solving, without trying to acquire any knowledge of the system on which to perform the tasks. This strategy was applied by 25% of the participants.

We compared the performance of the participants applying the two strategies with respect to the correctness of solution, use of documentation, use of compilation and execution facilities and the solution time. The assessment of the solutions showed that there was a correlation between modifying Task 2 successfully and using the systematic strategy. However, in Task 3 and Task 4 those who used the as-needed strategy also performed well. The participants who used the systematic strategy spent more time solving Task 2, but for Task 3 and Task 4 the results were reversed.

Three levels of programming expertise were identified and from the total number of 24 participants, eight participants were *object-oriented novice*, four participants were *procedural expert* and twelve participants were *object-oriented expert*. It was a clear difference in the comprehension approaches between the three groups of participants. The as-needed strategy was applied most by the object-oriented novices who also compiled more. These participants showed weaker results both in Task 2 and Task 4. Thus, novices should learn from the experts in order to maintain a system successfully.

Acknowledgements

First of all, I would like to thank my supervisor Amela Karahasanović for her encouragement, guidance and support with this research. Her feedback and advices have been very valuable to me. I am grateful to Simula Research Laboratory for support and great facilities. I thank Richard Thomas for providing database code and documentation of great value and Gunnar J. Carelius for his technical support. I am thankful to the participants in the experiment and Rolf Vassdokken and Johan Almqvist for preparing and testing the experiment material. I also thank the Student Association at Simula Research Laboratory and Nina Elisabeth Holt for social activities and lunches during the research period.

I specially thank Gøril Tømmerberg for being such a fantastic co-worker during our graduate studies and with this research. I also thank Jørgen Busvold for research collaboration and Line Joakimsen and Kristrún Arnarsdottir for inspiration and excellent teamwork in several courses.

Finally, I am deeply grateful to my parents for always supporting me and believing in me throughout my studies.

Oslo, May 2006
Kaja Kværn

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Software maintenance is widely recognised as an important part of a programmer's work. A central activity in software maintenance is program comprehension (Koenemann and Robertson 1991; Corritore and Wiedenbeck 2000; Storey et al. 2000). This is the process of understanding program code unfamiliar to the programmer, which is a non-trivial task. Some estimates put the cost of understanding software at 50% of the maintenance effort (von Mayrhauser and Vans 1997). Large programs hide a tremendous amount of complexity, and any attempt to modify them without understanding the relationships between components can introduce errors in the system. Thus, identifying how a programmer comprehends a program is very useful both for the development of techniques and in education. Bennedsen and Caspersen (2004) say that it is important that students gain insights into how programmers develop their solutions from the initial problems, e.g. how one frequently compiles code, use documentation and test partial solutions. One way of attaining this goal is to expose the students to how an expert programmer works.

Experience is believed to have a large impact on the quality and productivity of software development and maintenance work. Jones (1998) claims that staff experience are one of two factors with strongest impact on maintenance productivity. Nanja and Cook (1987) have showed that individual differences such as expertise can affect program comprehension strategies and performance and Burkhardt et al. (2002) point out that an important issue in program understanding is the effect of expertise on comprehension. Pennington (1987) suggests a difference in the mental representations of professional programmers of different levels of proficiency.

Program comprehension is the process of acquiring knowledge about a computer program (Rugaber 1995). But what does it actually mean to understand a computer program? Do you have to understand all the objects, all the methods and so on? Is it important to understand the overall behaviour of the program or only the behaviour of specific parts? Littman et al. (1986) say that to understand a program, a programmer must be familiar with the objects that the program manipulates and the actions that the program performs. According to Biggerstaff et al. (1994), one understands a computer program when one is able to explain different aspects of the program in a way that is qualitatively different from the way by which these aspects are expressed by the source code. Corritore and Wiedenbeck (2001) point out that program comprehension concerns an individual programmer's understanding of "what a program does and how it does it in order to make functional modifications and extensions to a program without introducing errors".

Numerous studies have been performed in the area of program comprehension. General research into software comprehension models suggests that the direction of comprehension is bottom-up (Pennington 1987), top-down (Brooks 1983) or a mixture of these two approaches (von Mayrhauser and Vans 1994). The breadth of comprehension is described by two strategies; systematic and as-needed (Littman et al. 1986). Research has also been conducted in the context of software maintenance (Koenemann and Robertson 1991; Visaggio 1999; Corritore and Wiedenbeck 2001; Parkin 2004), debugging (Nanja and Cook 1987; Ko and

Uttl 2003), design activities (Pennington et al. 1995) and effect of expertise (Nanja and Cook 1987; Burkhardt et al. 2002; Arisholm and Sjøberg 2004).

Although much research has been done, Corritore and Wiedenbeck (2001) point out that our understanding of program comprehension is incomplete and that there is a need for deeper knowledge on comprehension strategies during software maintenance. The study reported in this master thesis aim to add to the body of knowledge on program comprehension by identifying the comprehension strategies of professional developers and by exploring effects of strategies on performance and the effects of expertise on strategies and performance.

We have conducted a controlled experiment with 24 professional developers with different experience performing maintenance tasks on a previously unknown 3600 LOC Java application. The experimental design allowed the participants to choose how they wanted to comprehend the program or whether they wanted to solve the tasks directly. The participants had full Internet access and all available documentation and source code were presented online only. The JBuilder IDE provided compilation and execution facilities. Several previous studies have lacked these facilities (Koenemann and Robertson 1991; O'Brien et al. 2004; Parkin 2004) and no one has studied program execution in addition with use of documentation to determine the breadth of comprehension (Littman et al. 1986; Koenemann and Robertson 1991). In addition, the program used in this experiment was much larger than those in previous experiments (Littman et al. 1986; Koenemann and Robertson 1991; Burkhardt et al. 2002; Parkin 2004) and with professional developers instead of students (Karahasanovic et al. 2005).

All of the participants' actions were logged, and we have developed a data analysis tool in order to see their use of documentation, source code and compilation and execution facilities. In addition, feedback was collected through the feedback-collection method. The combination of these two sources of data gave us an opportunity not only to recreate the participants' actions on a very detailed level, but also to find about more why the participants chose a certain strategy.

1.2 Objective

The goal of the experiment was to examine the following:
1. Problem and problem solving strategies in maintenance of object-oriented systems
2. Comprehension-related activities during maintenance of object-oriented systems
3. Effects of strategies and expertise on program comprehension in maintenance of object-oriented systems

The first two topics are addressed by Jørgen Busvold and Gøril Tømmerberg in their theses (Busvold 2006; Tømmerberg 2006) respectively where more details on these topics can be found. The third topic is the focus of this thesis and the objective of this research was (1) to identify the comprehension strategies of professional developers and (2) to identify the effects of strategies on performance and the effects of expertise on strategies and performance.

A controlled experiment was conducted to address these issues. The collected data was used to address the following research questions:

- How do professional programmers comprehend an unknown application in order to perform maintenance tasks? Do they read the system documentation and test the application before they perform the tasks or do they go straight to the task solving without getting an overview of the application?
- What effect does the expertise have on the participants' comprehension strategies?
- What effects do the expertise and comprehension strategy have on the participants' task solution time and correctness of solution?
- What effects do the expertise and comprehension strategy have on the participants' use of documentation and compilation and execution facilities?

1.3 Research Method

We have performed a controlled software experiment with 24 participants. The participants were professional developers with different programming experience from five consultant companies. The participants were introduced to a Java application system of 3600 LOC, 27 classes, on which they should perform three maintenance tasks. Their actions were logged and feedback was collected through the feedback-collection method. The quality of the solutions was graded by an external consultant. Post-experimental group interviews were also performed.

1.4 Research Context

This master thesis is part of the Comprehensive Object-Oriented Learning (COOL) project. COOL is a 3-year research project launched in 2002 by a consortium of four Norwegian institutions: InterMedia, Norwegian Computing Center, Simula Research Laboratory and Department of Informatics at the University of Oslo.

COOL aims at gaining insights into the complex area of learning and teaching object-oriented concepts and raising awareness of the problem areas in the communities of Computer Science Education and Computer-Supported Collaborative Learning. More details about COOL can be found on the project's website (InterMedia 2006).

1.5 Contribution

There are several contributions of this research:

1. The further development of GRUMPS
2. Identification of professional developers' comprehension strategies
3. Effects of comprehension strategies on performance
4. Effects of expertise on strategies and performance

The further development of GRUMPS
A contribution of this research is the further development of the Generic Remote Usage Measurement Production System (GRUMPS) software (Evans et al. 2003). The low-level data is a rich source of information on the participants' actions during the study. We have

cleaned and extracted this data to get detailed information of each participant's actions during the tasks. The extraction of this data gives a detailed picture of the use of documentation, compilation and execution in addition to source code. The documentation and SQL-code written can be reused by researchers in similar studies and contribute to improve the understanding of developers' program comprehension strategies.

Identification of professional developers' comprehension strategies
This research has provided knowledge concerning program comprehension strategies of professional programmers. Two different approaches to the task solving were identified. In the first one, called systematic strategy, the participants tried to get an overview of the application before performing the maintenance tasks, by reading the documentation available or running the application. 75% of the participants applied this strategy. In the second one, called as-needed strategy, the participants went straight to the task solving without trying to first understand the system. This strategy was applied by 25% of the participants. This result differs from other studies in that no one has found that so many of the participants applied the systematic strategy. Thus the identification of professional programmers' comprehension strategies extends today's knowledge of program comprehension.

Effects of comprehension strategies on performance
Another contribution of this research is how the strategies used by the participants affected their performance. The assessment of the solutions showed that there was a correlation between modifying the first task successfully and using the systematic strategy. However, in Task 3 and Task 4 those who used the as-needed strategy also performed well. The participants who used the systematic strategy spent more time performing Task 2, but for Task 3 and Task 4 the results were reversed. The findings are valuable in that software engineers get insight in how to comprehend an unknown application in order to perform maintenance tasks correctly and most efficiently. Programmers should be aware of the different strategies and know when to apply them successfully.

Effects of expertise on strategies and performance
In addition to the investigation of effects of strategies on program comprehension, this research has explored the effects of expertise on strategies and performance. It was a clear difference in the comprehension approaches between the three groups of participants. The object-oriented novices applied the as-needed approach more than the other participants. Similar results have been found in previous research. Hence this study confirms and extends today's knowledge of effect of expertise on program comprehension strategies. The object-oriented novices had a low correctness percentage both in Task 2 and Task 4. Thus, novices should learn from the experts in order to maintain a system successfully. This research has contributed to this by showing how professional programmers use documentation, source code and how they compile and execute the application. This knowledge can be used to improve education and in the development of techniques.

1.6 Chapter Overview

Parts of this thesis have been written in collaboration with two other MSc. students, Jørgen Busvold (2006) and Gøril Tømmerberg (2006). These parts are thus common for all three theses. This experiment is a replication of the experiment described by Levine (2005) and thus some sections have been taken from her thesis. The reminder of the thesis is organized as follows:

Chapter 2
Related work

This chapter presents an overview of the related work. The identification of related work on program comprehension has been performed in collaboration with Busvold (2006) and Tømmerberg (2006). Section 2.2 gives an overview of existing comprehension models and section 2.3 presents relevant empirical studies of program comprehension. These sections are quite similar in all three theses. The empirical studies of effect of expertise on program comprehension in section 2.4 are presented in this thesis only.

Chapter 3
Method

This chapter gives a detailed description of the controlled software experiment. The experiment is designed by Karahasanović (2005). The description of the experiment, tasks and application are mainly based on Levine (2005). The further development of GRUMPS is described in section 3.3.1 and by Tømmerberg (2006). All sections except 3.3.1 are common for all three theses.

Chapter 4
Analysis

This chapter describes the analysis of the data collected by GRUMPS and the feedback-collection, including data preparation. The data preparation for the analysis, correctness and solution time in section 4.1 are common for Tømmerberg (2006) and this thesis. The classification of developers in section 4.1.4 and the identification of comprehension strategies in section 4.2 are presented in this thesis only.

Chapter 5
Results

The results of the study are presented and discussed in this chapter. Section 5.1 presents the results regarding comprehension strategies. Section 5.2 presents the results on effects of strategies on performance whereas section 5.3 presents the results on effect of expertise on strategies and performance. Finally, section 5.4 summarizes the results.

Chapter 6
Validity

The most important threats to the validity of the experiment are discussed in this chapter. Some of the threats mentioned can also be found in Tømmerberg (2006), but there they are specially related to her research.

Chapter 7 Conclusions and Future Work	This chapter summarizes the contributions of the research and suggests implications for future work.
Appendix A List of Strategies and Developer Classification	The appendix gives a detailed overview of the participants’ strategies and expertise.
Appendix B Background Questionnaire	Questionnaire used to state the participants’ background: their education, programming experience and experience of tools.
Appendix C Tasks and Detailed Task Description	The appendix contains the maintenance tasks the participants performed in the controlled experiment and also describes how the tasks can be solved in detail. The descriptions are written by Levine (2005) and are common for all three theses.
Appendix D The Application	The application is described in this appendix. The description is written by Levine (2005) and is common for all three theses.

2 Related Work

This chapter gives an overview of previous research conducted in the field of program comprehension. The identification of related work on program comprehension in section 2.1 has been performed in collaboration with Busvold (2006) and Tømmerberg (2006). Section 2.2 gives an overview of existing comprehension models and section 2.3 presents relevant empirical studies of program comprehension. These sections are quite similar in all three theses. The empirical studies of effect of expertise on program comprehension in section 2.4 are presented in this thesis only.

2.1 Identification of Related Work

The identification of related work has been conducted in two separate parts. The first part is identification of related work on program comprehension. This work has been done in collaboration with Busvold (2006) and Tømmerberg (2006). We performed the searches individually and merged the results. In order to find related work we have searched several digital libraries and reference databases: ACM Digital Library, IEEE Xplore, ISI Web of Knowledge, INSPEC and DUO. We also used the search engines Google and Google Scholar. In addition to this I have performed searches on my own to identify work related to this research only that is the effects of expertise on program comprehension.

The keywords used during the search have been combined in different expressions and are the following:

- program comprehension
- software comprehension
- strategies
- experiment
- object-oriented
- software maintenance
- expertise
- experience
- novice
- expert

The search was initially performed in November 2005 and repeated in March 2006. The initial search on "program comprehension" gave about 500 results. By adding more keywords to the phrase, the number of results was considerably reduced. I filtered out the irrelevant articles, conference proceedings and theses by first reading the title and then the abstract. I have also used references in relevant articles to find other related articles. This work resulted in 32 articles and conference proceedings closely related to this research.

2.2 Program Comprehension Strategies

According to von Mayrhauser and Vans (1996) a comprehension strategy is the high-level or overall approach to comprehension. It is a style or method of understanding. A strategy guides the sequence of actions while following a plan to reach a particular goal. The topics on program comprehension that have been studied mostly are the direction of comprehension and the breadth of comprehension. The direction of comprehension activities concerns whether the strategic approach to program comprehension is top-down, bottom-up, or a mixture of the two. The breadth of comprehension activities refers to the breadth of familiarity with the program gained by the programmer during comprehension activities (Corritore and Wiedenbeck 2000).

2.2.1 Direction of Comprehension

The top-down model of program comprehension was first proposed by Brooks (1983). The programmer starts with a general hypothesis about what the program does, and the components are then viewed in light of this hypothesis. The top-down model is typically invoked if the code or type of code is familiar.

The bottom-up model, as described by Pennington (1987), consists of two program abstractions formed by the programmer during comprehension. The program model is a low-level abstraction and is formed early during program understanding using information in the program text. The domain model is a higher-level abstraction containing knowledge of data flow and function. This abstraction is formed after the program model.

While the top-down and bottom-up models have been very influential, today mixed models of program comprehension are increasingly viewed as more realistic descriptions of large program comprehension (Corritore and Wiedenbeck 2000). von Mayrhauser and Vans (1994) observed that program understanding involves both top-down and bottom-up activities, and this led to the formulation of a model that includes the following existing models as components: (1) program model, (2) situation model, (3) domain model and (4) knowledge base. The integrated model considers programmers to behave opportunistically in program understanding, switching from top-down to bottom-up comprehension strategies depending on the situation. von Mayrhauser and Vans propose that programmers use a top-down approach to understanding when they are working in a familiar domain. On the other hand, a bottom-up approach is used when the code and domain are unfamiliar.

2.2.2 Breadth of Comprehension

Littman et al. (1986) have identified two strategies concerning breadth of comprehension, the systematic strategy and the as-needed strategy. These two strategies are defined within the context of comprehending an entire program; hence the systematic approach attempts to comprehend the program as a whole before any attempt at maintenance is made. On the other hand, using the as-needed strategy, the programmer attempts to minimize the amount of code that has to be understood. The programmer does not attempt to understand the overall design of the program but concentrates instead on the functioning of selected local parts of the code that are critically involved in the modification.

Conclusions which are drawn from the use of these strategies are that the systematic approach requires more time during the comprehension stage of maintenance but results in a strong cognitive model of the program. In contrast, the as-needed approach requires less comprehension time but will result in a weaker and incomplete cognitive model (Young 1996).

Littman et al. (1986) claim that a systematic strategy plays an important role for comprehension. However, Koenemann and Robertson (1991) found that the programmers in their experiment did not attempt to use a truly systematic strategy. The programmers did not spend time comprehending parts of the program that they believed irrelevant to the modification task. Koenemann and Robertson argue that these differences are caused by the fact that their program was significantly larger in size. They carried out an experiment on a 636 LOC program, while Littman et al. used a program of only 250 LOC.

von Mayrhauser and Vans (1996; 1997) also argue that the systematic strategy is unrealistic for large programs, even though it seems better or safer. However they observed an instance of systematic study of large program (von Mayrhauser and Vans 1994). Both Littman et al. (1986) and von Mayrhauser et al. agree that a disadvantage to the opportunistic approach is that understanding is incomplete and code modifications based on this understanding may be error prone.

Karahasanović et al. (2005) conducted an experiment to explore the claim that in large program it is unrealistic to acquire knowledge of how a program works before modifying it. Thirty-eight students in their third or fourth year of study in computer science performed several maintenance tasks on a medium-size Java application system in a six-hour long experiment. The results showed that the subjects who applied the systematic strategy performed better. Two major groups of difficulties were related to the comprehension of the application structure and to the inheritance of functionality. The results indicated that the first group of difficulties can be alleviated by applying the systematic strategy.

2.3 Empirical Studies of Comprehension Strategies

Several experiments have been conducted in order to determine programmers' comprehension strategies, both in the procedural and object-oriented paradigm. Table 1 and 2 summarizes studies regarding participants, applications (language and size), tasks (type of task and duration), environments (e.g. hardcopy or computer), data collection method and purpose of the study. The experiments which have been conducted in both paradigms are only listed once under the object-oriented paradigm (Table 1).

Table 1: Empirical studies of comprehension strategies in the object-oriented paradigm

Study	Participants	Application Task Environment	Data Collection Method	Purpose
Ramalingam and Wiedenbeck (1997)	75 students	C++ 6 short programs Hard copy 3 object-oriented 3 not object-oriented (written in C) 5 comprehension questions for each program	Demographic questionnaire Questionnaire	Determine whether the mental representation of object-oriented programs differs from imperative programs for novice programmers
Burkhardt et al. (1998)	49 professionals (28 experts, 21 novices)	C++ 550 LOC Comprehension for later documentation or reuse. 35 min. program study.	Verbal protocols Questionnaire	Analyze OO program comprehension and examine the effects of expertise 3 directions
Wiedenbeck et al. (1999)	86 subjects (half were novices in the OO style while half were novices in the procedural style)	Pascal C++ Three short programs One longer program 15 – 25 LOC 150 LOC Hard copy	Questionnaire Demographic questionnaire	Compare mental representations and program comprehension by novices in the object-oriented and procedural styles
Corritore and Wiedenbeck (2000; 2001)	30 professionals (15 OO experts, 15 procedural experts)	C, C++ 783 and 822 LOC Maintenance tasks Two 3-hour sessions (7 to 10 days apart).	Screen capture software	Program understanding strategies employed during comprehension and maintenance activities carried out over time
Burkhardt et al. (2002)	51 subjects (30 experts, 21 novices)	C++ 550 LOC Library problem Documentation or reuse tasks	Verbal protocols Questionnaire	Evaluate the effect on program comprehension of three factors: programmer expertise, programming task and the development of understanding over time
Torchiano (2004)	28 students (4th year)	Java 628 LOC Maintenance tasks Code and documentation available on-line	User action capture software	Non-intrusive approach to study comprehension cognitive models
Blinman and Cockburn (2005)	11 postgraduate students	MS.NET J# 4 small console applications Hard copy 10-20 LOC	Multiple Choice Questions	How comprehension is influenced by naming style and documentation
Karahasanović et al. (2005)	39 students	Java 3600 LOC	User action recorder	Comprehension strategies and

		Three maintenance tasks	Verbal protocols	difficulties in maintenance of object-oriented systems
Ko et al. (2005)	10 experts	Java 503 LOC in 9 classes Eclipse IDE 5 maintenance tasks 70 minutes	Screen capture videos	Discover fundamental activities in maintenance work and use this understanding to elicit design requirements for new tools to support maintenance tasks

Table 2: Empirical studies of comprehension strategies in the procedural paradigm

Study	**Participants**	**Application Task Environmen**	**Data Collection Method**	**Purpose**
Woodfield et al. (1981)	48 professionals	Fortran 8 versions of the same program Answer questions related to the programs	Questionnaire	How different types of modularization and comments are related to programmers' ability to understand programs
Littman et al. (1986)	10 professionals	Fortran 250 LOC Maintenance	Videotapes	Scope of comprehension: systematic and as-needed
Koenemann and Robertson (1991)	12 professionals	Pascal 636 LOC 4 maintenance tasks 15 – 44 min spent on modification Documentation available	Verbal protocols Audiotape	Scope of comprehension: systematic and as-needed
von Mayrhauser and Vans (1994)	11 professionals	Maintenance	Verbal protocols	Find a code comprehension process model
Shaft (1995)	24 professionals	COBOL Accounting (familiar domain) 417 LOC Hydrology (unknown domain) 416 LOC Comprehension questions	Verbal protocols Questionnaire	Comprehension strategy – use of metacognition
Ye and Salvendy (1996)	20 students (10 novices and 10 intermediate)	C Sheets of program code and plan hierarchy The task was to match each program segment with its goal	Monitoring (recording and timing)	Conducted to examine skill differences in the control strategy for computer program comprehension
von Mayrhauser and Vans (1996)	11 professionals	Pascal Large scale production	Verbal protocols (audio-taped)	Find a code comprehension

		code. Modules from ≤200 to ≥9000 LOC Maintenance task Two hours session		process model using the Integrated Comprehension model as a guide for large-scale program understanding
von Mayrhauser and Vans (1997)	4 professionals	Pascal Min 40 000 LOC Corrective maintenance tasks Observational field study Two hours session	Verbal protocols (audio and/or video taped)	The paper reports on the general understanding process, the types of actions programmers preferred during the debugging task, and the level of abstraction at which they were working
Visaggio (1999)	1. & 2. exp.: undergraduate students 3. exp: professionals	Pascal Maintenance		Compares the maintenance process from two paradigm: Quick Fix & Iterative Enhancement
Storey et al. (2000)	30 students (5 graduate and 25 senior undergraduate)	C A 2-h session orientation (5 min), training tasks (20 min), practice tasks 300 LOC in 12 files (20 min), formal tasks 1700 LOC in 17 files maintenance (50 min)	Background questionnaire Verbal protocols Video-tape Post-study questionnaire Post-study interview	Explore whether program understanding tools enhance or change the way that programmers understand programs
O'Brien et al. (2004)	8 professionals	COBOL Two programs each 1200 LOC Hardcopy of source code Three sessions (1 practice and 2 experimental) 1.5 hour	Verbal protocols	Distinguish between two comprehension processes that have previously been grouped together as 'top-down'
Parkin (2004)	29 students (3 or more years into Computer Science studies)	C 281 LOC Maintenance (enhancement and correction) 1.5 hour	Demographic data Logging Verbal protocols (audio-taped)	Program comprehension strategies employed during maintenance tasks
Fitzgerald et al. (2005)	Students	Multiple Choice Questions	Verbal protocols	Analyze code comprehension strategy

Purpose of study

The studies were in the area of program comprehension and examined how various conditions affected the comprehension and strategies. They have examined the effect of expertise in the object-oriented paradigm (Burkhardt et al. 1998; Burkhardt et al. 2002) and the procedural paradigm (Ye and Salvendy 1996), software maintenance and enhancements in both paradigms (Corritore and Wiedenbeck 2001), object-oriented paradigm (Karahasanovic et al. 2005; Ko et al. 2005) and procedural paradigm (Littman et al. 1986; Koenemann and Robertson 1991; Visaggio 1999; Parkin 2004), effects of phase (Corritore and Wiedenbeck 2000; Burkhardt et al. 2002), use of documentation by procedural programmers (Parkin 2004) and object-oriented programmers (Torchiano 2004; Blinman and Cockburn 2005), effects of modularization and comments (Woodfield et al. 1981), domain familiarity (Shaft 1995; O'Brien et al. 2004), effects of programming paradigm (Ramalingam and Wiedenbeck 1997; Wiedenbeck et al. 1999; Corritore and Wiedenbeck 2000; Corritore and Wiedenbeck 2001), effects of tools (Storey et al. 2000), large scale code (von Mayrhauser and Vans 1994; von Mayrhauser and Vans 1996; von Mayrhauser and Vans 1997) and tasks (von Mayrhauser and Vans 1997; Burkhardt et al. 2002; Parkin 2004).

Data Collection Method

The data were collected by means of verbal protocols (Koenemann and Robertson 1991; von Mayrhauser and Vans 1994; Shaft 1995; von Mayrhauser and Vans 1996; Burkhardt et al. 1998; Storey et al. 2000; Burkhardt et al. 2002; O'Brien et al. 2004; Parkin 2004; Fitzgerald et al. 2005; Karahasanovic et al. 2005), screen capture software (Corritore and Wiedenbeck 2000; Corritore and Wiedenbeck 2001; Ko et al. 2005), user action recorder (Torchiano 2004; Karahasanovic et al. 2005) and questionnaire (Woodfield et al. 1981; Shaft 1995; Ramalingam and Wiedenbeck 1997; Burkhardt et al. 1998; Wiedenbeck et al. 1999; Storey et al. 2000; Burkhardt et al. 2002; Blinman and Cockburn 2005).

Identification of Strategy

The comprehension strategies were identified by the proportion of documentation and code files accessed (Burkhardt et al. 1998; Corritore and Wiedenbeck 2000), percentage of program parts and lines of code accessed (Koenemann and Robertson 1991), times and proportion of files accessed (Parkin 2004), number of web pages visited and the time spent on each page (Torchiano 2004), verbal statements and the components studied (Littman et al. 1986) and classification of verbal statements related to the integrated cognition model (von Mayrhauser and Vans 1994; von Mayrhauser and Vans 1996; von Mayrhauser and Vans 1997).

The direction of comprehension has been identified in several ways. The top-down approach has been characterized with that the programmers access files at the most abstract level (documentation and program header information) and the bottom-up approach by accessing the low-level implementation files (Burkhardt et al. 1998; Corritore and Wiedenbeck 2000; Corritore and Wiedenbeck 2001; Parkin 2004). The top-down approach has also been identified by assessment of global description (Koenemann and Robertson 1991) or by a small number of web pages visited and a long time spent on each page (Torchiano 2004). Torchiano defined the bottom-up approach as spending short time on a large number of pages. von Mayrhauser and Vans (1994; 1996; 1997) identified the direction of comprehension by classifying the actions types related to the domain (top-down), situation or program model.

Littman et al. (1986) described the systematic strategy as reading all the code to understand the global program behavior before making any changes. The programmer using the as-needed strategy attempts to minimize studying the program to be modified. The systematic breadth of comprehension has also been indicated by a broad study of all material while the as-needed was categorized by limitations of the study of program materials to only a small part of those available (Koenemann and Robertson 1991; Corritore and Wiedenbeck 2000; Corritore and Wiedenbeck 2001) or by reading documentation, source code and executing the application prior to the task solving in order to get an overview (Karahasanovic et al. 2005).

Results
von Mayrhauser and Vans (1994; 1996) reported that code size affected the level of abstraction. Large size code increased participants work with higher level program details. They identified that programmers used a multilevel approach to understanding, frequently switching between program, situation and domain (top-down) models. This is supported by the results of Corritore and Wiedenbeck (2000; 2001) who found that the object-oriented participants initially used a top-down approach by focusing strongly on documentation. Later their strategy shifted to a more bottom-up orientation. The procedural participants employed a more bottom-up strategy during the study. In the study by Torchiano (2004) the result showed no clear separation among the top-down and the bottom-up behaviours. Torchiano argued that this agrees with the integrated cognitive model that predicts a switching between models.

Expertise in domain and programming language allowed more top-down comprehension, while little experience meant that comprehension was more bottom-up (von Mayrhauser and Vans 1997). Burkhardt et al. (1998) also found evidence of top-down behaviour in expert comprehension, but in the beginning the participants read files at both the top and bottom of the hierarchy showing both top-down and bottom-up direction. Ye and Salvendy's (1996) results indicated the use of an overall top-down strategy by both intermediates and novices for program comprehension but novices demonstrated more random behaviours and thus opportunistic strategies than intermediates.

The type of task also affected the comprehension. In adaptive maintenance the subjects used more top-down comprehension than in corrective (von Mayrhauser and Vans 1997). The results from Parkin's study (2004) showed that programmers doing corrective tasks utilized documentation and header information and thus a more top-down approach than programmers undertaking an enhancement. Enhancers made more specific use of task documentation than corrective programmers and switched from initial top-down processing to bottom-up earlier.

In Littman et al.'s study (1986) all programmers acquired the static knowledge about the program, but only the programmers who used the systematic strategy acquired the necessary causal knowledge. The programmers using the systematic approach constructed more successful modifications than those using the as-needed approach. Karahasanović et al. (2005) also found that the subjects who applied the systematic strategy performed better than those who used the as-needed strategy. Koenemann and Robertson (1991) reported a very restricted scope of comprehension. In this study none of the participants used a systematic strategy and the authors argued that the systematic approach was unrealistic in large program. Corritore and Wiedenbeck (2000; 2001) reported that the procedural programmers employed a wider scope of comprehension than the object-oriented throughout the experiment. The scope of both groups was wider in the beginning and then narrowed.

Shaft (1995) found that programmers use metacognition when studying computer programs. Two-thirds of the programmers indicated that they purposefully choose a comprehension strategy. When studying a computer program from an unfamiliar application domain, however, programmers who generally use metacognition had lower levels of comprehension than when working in a familiar domain or when compared to those programmers who did not use metacognition.

Fitzgerald (2005) identified 19 different strategies when students answered code-based multiple choice questions. All the students employed a range of strategies and often multiple strategies on each individual problem. They also applied different strategies to different types of questions, and often applied the strategies poorly.

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 (Blinman and Cockburn 2005). Woodfield et al. (1981) also found that those subjects whose programs contained comments were able to answer more questions than those without comments.

Storey et al. (2000) investigated how three tools aided the comprehensions process and noted that the tools did enhance the users' preferred comprehension strategies while solving the tasks. For example, the ability to view dependency relationships in all three tools was exploited by most of the users and the ability to switch seamlessly between high-level views and source code was considered a desirable feature. However, the tools hindered the users' progress in some instances. The lack of an effective source code searching tool in two of the tools caused some users to change their comprehension approach for some of the tasks. In one of the tools, insufficient high-level information forced some users to adopt a more bottom-up approach for understanding. Ko et al. (2005) identified several opportunities for new tools that could save programmers a lot of work.

2.4 Empirical Studies of Effect of Expertise on Program Comprehension

Several studies have been conducted in the area of program comprehension, software maintenance and effect of experitse. Table 3 gives an overview of studies regarding participant categories, applications (language and size), tasks (type of task and duration), environments (e.g. hardcopy or computer), data collection method and purpose of the study.

Table 3: Empirical studies of effect of expertise on program comprehension

Study	Participant categories	Application Task Environment	Data Collection Method	Purpose
Nanja and Cook (1987)	Novice Intermediate Expert	Pascal program 73 LOC Debug three semantic and three logics errors	Verbal protocols	Comparing the debugging process of expert, intermediate and novice student programmers
Pennington et al. (1995)	Expert procedural designer Expert OO designer Novice OO designer & Expert Procedural designer	Scoring system for swim meet competitions Design activities	Verbal protocols	Provide descriptions of design activities and of the evolving designs for expert procedural and expert object-oriented (OO) designers and for novice OO designers who also had extensive procedural experience
Ye and Salvendy (1996)	Novice Intermediate	C Sheets of program code and plan hierarchy The task was to match each program segment with its goal	Monitoring (recording and timing)	Conducted to examine skill differences in the control strategy for computer program comprehension
Burkhardt et al. (1998)	Novice Expert	C++ 550 LOC Comprehension for later documentation or reuse. 35 min. program study	Verbal protocols Questionnaire	Analyze OO program comprehension and examine the effects of expertise 3 directions
Davies (2000)	Novice Expert	The participants were briefly shown 4 small programs written in C++. The participants were then asked to answer a series of comprehension questions	Questionnaire	Try and tap more directly the cognitive representations used by novice and expert programmers by presenting programs for a very short period of time
Burkhardt et al. (2002)	Novice Expert	C++ 550 LOC Library problem Documentation or reuse tasks	Verbal protocols Questionnaire	Evaluate the effect on program comprehension of three factors: programmer expertise, programming task and the development of understanding over time
Jørgensen and Sjøberg (2002)	Professionals Different measures to define experience	A random set of maintenance tasks was selected. The maintained applications were,	Verbal protocols	Examine the relationship between amount of experience and maintenance

		typically, written in COBOL, 4GL or C and varied in size from a few thousand to about 500 000 LOC		skills
Ko and Uttl (2003)	Students with different education	Debugging a simple program in an unfamiliar programming system - a programmable statistical package	Verbal protocols Screen capture software	Examine the effect of individual differences on the program comprehension strategies of users working with an unfamiliar programming system
Arisholm and Sjøberg (2004)	Undergraduate Graduate Juniors Intermediate Seniors	Several change tasks on two alternative Java designs	Questionnaire	Investigate the claims that a delegated control style represents object-oriented design at its best, whereas a centralized control style is reminiscent of a procedural solution, or a "bad" object-oriented design

Purpose of Study

The studies have examined the impact of expertise on maintenance skills (Jørgensen and Sjøberg 2002), task and phase (Burkhardt et al. 2002), cognitive representations (Davies 2000), design activities (Pennington et al. 1995), debugging (Nanja and Cook 1987), control strategy (Ye and Salvendy 1996), delegated versus centralized control style (Arisholm et al. 2001) and three dimensions of comprehension strategies (Burkhardt et al. 1998).

Data Collection Method

The data were collected by means of verbal protocols (Nanja and Cook 1987; Pennington et al. 1995; Burkhardt et al. 1998; Burkhardt et al. 2002; Jørgensen and Sjøberg 2002; Ko and Uttl 2003), questionnaire (Burkhardt et al. 1998; Davies 2000; Burkhardt et al. 2002; Arisholm and Sjøberg 2004), monitoring (Ye and Salvendy 1996) and screen capture software (Ko and Uttl 2003).

Participants

The number of participant categories used varies in the studies. Burkhardt et al. (1998; 2002), Davies (2000) and Ye and Salvendy (1996) have used two categories whereas Pennington et al. (1995) and Nanja and Cook (1987) have used three. Arisholm and Sjøberg (2004) used five developer categories in their research. Ko and Uttl (2003) and Jørgensen and Sjøberg (2002) have not used a fixed number of categories, but have used different measures to determine the subjects' expertise.

The participants' background also differs between the studies. Pennington et al. (1995) and Jørgensen and Sjøberg (2002) have used professional developers only, whereas Arisholm and Sjøberg (2004) have used both students and professionals. Burkhardt et al.s' (1998; 2002) experts were also professionals whereas Davies' (2000) experts were experienced C++

programmers who were either teaching this language or used it extensively in their research. Both Burkhardt et al. (1998; 2002), Davies (2000) and Nanja and Cook (1987) used undergraduate computer science students for their novice category, but as Davies (2000) used first year undergraduate computer science students, Burkhardt et al.'s (2002) were advanced undergraduate computer science students. Ye and Salvendy's (1996) participants were both graduate and undergraduate students and Nanja and Cook (1987) used students that were just finishing their second term of an introductory Pascal programming course. Ko and Uttl (2003) used students with different education.

Results

All of the studies found some differences between participant categories, but not for all conditions. Nanja and Cook (1987) found that only experts tried to understand the program first and that they also were the fastest and most successful in correction all of the errors. Ye and Salvendy (1996) identified the use of an overall top-down strategy by both intermediates and novices for program comprehension. Novices' control strategies involved more opportunistic elements than experts' in the overall top-down process of program comprehension. Those differences in the control strategy between intermediates and novices resulted in better performance in intermediates than novices.

Jørgensen and Sjøberg (2002) found that the most experienced maintainers did not predict maintenance problems better than maintainers with little or medium experience. More application specific experience, however, further reduced the frequency of major unexpected problems. Ko and Uttl (2003) also found that debugging performance depends on bug-specific domain knowledge.

Several differences between novices and experts in design activities have been found (Pennington et al. 1995). The novices did not analyze the problem situation through their objects and retained a few procedural features in their design. Novices also differed from both expert object-oriented and procedural designers by spending very little time evaluating the design. However, the novices had completeness scores that were only a few points below those of the object-oriented experts and above the procedural experts. The novices also showed the same trends for top-down, breadth-first expansion as did the experts.

Skilled developers require less time to maintain software with a delegated control style than with a centralized control style (Arisholm and Sjøberg 2004). Arisholm and Sjøberg (2004) found that more novice developers had serious problems understanding a delegated control style and performed far better with a centralized control style. Experts are also able to quickly and accurately extract object-oriented information from briefly presented object-oriented programs (Davies 2000). There is though no difference between different comprehension categories as presentation length increases.

Burkhardt et al. (1998) found that subjects were similar in the scope of their comprehension, although the experts tended to consult more files. They found strong evidence of top-down processes in expert comprehension. They also found evidence of execution-based guidance and less use of top-down processes in novice comprehension.

2.5 Summary

Numerous studies have been performed in the area of program comprehension both in the object-oriented and the procedural paradigm. Previous research has also focused on how the participants' expertise affects the program comprehension and performance. However, experiments on program comprehension have been criticised for their lack of realism. The programs used in the empirical studies found are typically small (Nanja and Cook 1987; Koenemann and Robertson 1991; Corritore and Wiedenbeck 2001; Burkhardt et al. 2002). Only one observational study of large-scale program comprehension in the industrial context has been reported (von Mayrhauser and Vans 1996; von Mayrhauser and Vans 1997). Several studies also did not allow the participants to compile and execute the program (Koenemann and Robertson 1991; Parkin 2004). Furthermore, no studies have determined the breadth of comprehension of professional programmers by using detailed information about how they read documentation and use program execution to familiarize with the system.

In this study I intend to extend previous research by identifying the comprehension strategies of professional programmers. The program used is larger in size than most programs used in previous studies. The effects of the strategies on performance and the effects of strategies and expertise on performance will also be investigated.

3 Method

This chapter gives a detailed description of the controlled software experiment. The experiment is designed by Karahasanović (2005). The description of the experiment, tasks and application are mainly based on Levine (2005). The further development of GRUMPS is presented in section 3.3.1 and also by Tømmerberg (2006). All sections except 3.3.1 are common for Busvold (2006), Tømmerberg (2006) and this thesis.

3.1 Participants

The participants in this experiment were 24 employees from five different software companies. The size of the companies ranged from small (five employees) to medium size (over 100 employees). All companies developed software products for both the private as well as the governmental sector. All of the firms offered consulting services though some of them reported that their main focus was product development. The application background specialties ranged widely from Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Systems Analysis and Program Development (SAP), advisory service, sale, and mobile technology to special development. The companies seem to be separated between focus on concrete methods like Rational Unified Process (RUP) on the one hand and more agile methods like Extreme Programming (XP) on the other. But there was also one company that claimed the method depended on their custom relationship.

All of the 24 participants were male, and the mean age was 31.9 range 25 to 50. They had taken between 0 to 80 credits (a full school year is 20 credits) in programming courses, the median being 30. Their programming experience ranged from 0 to 25 years and the median Java work experience was 2.35 years range 0 to 8. The participants self-reported they had produced between 1000 to 1000000 lines of Java code (median 40000). Almost all of the participants self-reported their skills in Java-programming as medium to good (median 4 on a five-point scale: 1 equals no experience with Java, 5 means expert). Their knowledge of JBuilder was self-reported as below medium (median 2 on a five-point scale: 1 equals no experience with JBuilder, 5 equals expert).

All of the developers were paid for their participation in the experiment.

3.2 Experimental Procedure

A pre-test with two MSc. students were undertaken to ensure that the tasks, documentation and tools were appropriate.

The main study consisted of three phases. In the first phase, a week before the experiment, the participants filled in the background questionnaire answering questions on their education, programming experience and experience with tools via a web-based tool. Usernames and password needed to answer questionnaires were distributed via e-mail. The second phase was the experiment. First, participants were welcomed, informed about the experiment's goals and procedure. They were also informed that all their actions were logged by a logging tool. Second, the participants were asked to sign a consent form. They agreed that they would not share information about the experiment with their colleagues, either during or after the experiment. The researchers guaranteed confidentiality and anonymity. After the introduction, the participants were given a unique username and password to be used during the day. Then, all participants were asked to solve a small training task and a pre-test calibration task. After that the participants were asked to conduct three maintenance tasks on the library application while writing feedback every 15 minutes on a screen. There was a 30 minutes lunch break during the experiment and the participants were allowed to take shorter (few minutes) breaks when needed.

The experiment was conducted in five separate sessions on five separate days. The participants could choose the day they wanted to participate in the experiment. The number of participants per session was five on four days and four on one day. The participants were accommodated in a laboratory with one observer and one for technical assistance. Post-experimental group interviews were conducted by one interviewer and one observer.

3.3 Tools and Data Collection

A Web-based tool, the Simula Experiment Support Environment (SESE) (Arisholm et al. 2002) was used for logistics support. The participants used SESE throughout the experiment to answer the background questionnaire, download documents and source code, upload their solutions and provide feedback. Start-time and end-time for each task were also recorded by SESE.

Keystrokes, mouse-clicks and window focus events were logged with timestamps in milliseconds by the GRUMPS-Lite software (Thomas et al. 2003). JBuilder 9 (Borland Software Corporation) was used as the programming environment for the experiment.

3.3.1 GRUMPS

The Generic Remote Usage Measurement Production System (GRUMPS) is developed at Glasgow University (Evans et al. 2003). The goal of GRUMPS is to provide general purpose mechanisms for the capture of user actions for subsequent analysis and mining. In this experiment we used a reliable, low complexity version called GRUMPS-Lite. It includes a User Action Recorder (UAR) that runs under Windows. It can monitor all window activation, mouse and keyboard events, but can be restricted by the user or investigator when required.

There is a transport mechanism to harvest collected data and store it in the repository, a SQL Server database. The repository is designed for flexibility and is extremely simple. The database consists of three main tables as shown in Figure 1(a) which is taken from Thomas and Mancy (2004). These tables are the basis of the data cleaning process. The XML fields store tagged data appropriate to the circumstances, such as shown in the Figure 1(b) for a Change Window Focus event.

The repository design has proved to be very robust, and well adapted for rapid collection of large volumes of data. Thus, collecting large volumes of low level data has become technically feasible.

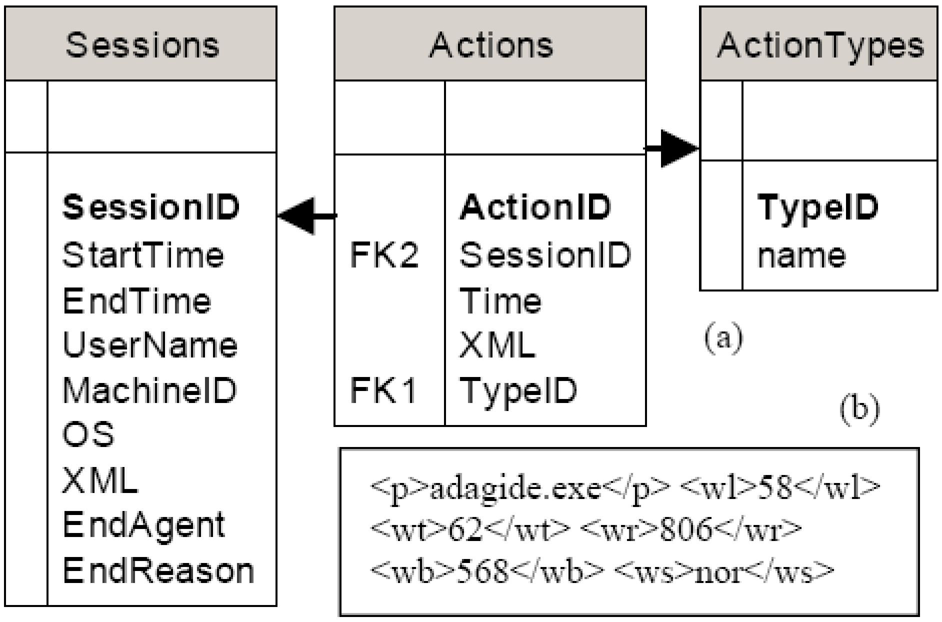


Figure 1: (a) The repository schema. (b) XML for a change of window focus event.

Difficulties Experienced in Previous Studies

Several studies report that the data preparation for the analysis of low-level data is an extremely difficult process. A number of researchers have noted that generic data collection often generates unwieldy and unmanageable data from which it is difficult to extract meaningful information (Misanchuk and Schwier 1992; Reeves and Hedberg 2003).

Renaud and Gray (2004) have analyzed data collected by GRUMPS. They report on a method of data cleaning and analytical preparation of low-level usage data which can be used in similar studies. In particular they discuss the particular challenges confronted during a study based on low-level keystroke data. They point out that cleaning and meaningfully interpretation of low-level data is not an easy task.

Thomas, Kenney et al. (2003) report from a study that investigated the use and usefulness of the GRUMPS software. They claim that the data preparation phase of the investigation is a large bottleneck and experienced that the data preparation took about a full person-month.

Our Work with the Database

The creation of the full database from the raw GRUMPS logs (Session, Actions, Actiontypes) was done by Richard Thomas, and the steps performed are described in a working draft written by him. We have developed this database further based on Thomas' documentation and previous SQL code written. Our goal was to get further knowledge of the participants' comprehension related activities during the study. We especially wanted to examine their use of documentation and program execution behaviour. The database already contained two tables that held information on the subjects' use of Java classes, Class and ClassActivity. We used a similar structure as in these tables. Our new tables are described in Table 4.

Table 4: Overview of new tables created

Table	Content
Doc	Documents and web pages.
DocumentActivity	Activities related to documents and web pages.
Category	Categorization of the activities
Library	Window titles related to compilation and execution of the application.
LibraryActivity	Activities related to compilation and execution of the application.

In order to make the tables described in Table 4 and extract the data for analysis, we went through a process that is illustrated in Figure 2. The steps performed by us are marked in grey.

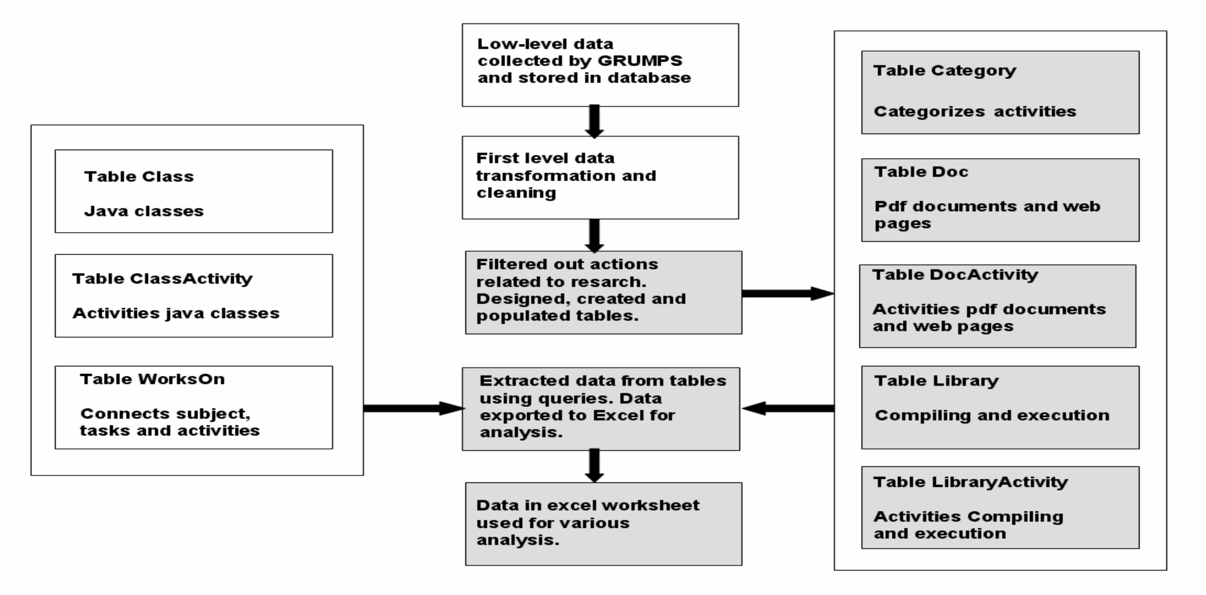

Low-level data collected by GRUMPS and stored in database
First level data transformation and cleaning
Filtered out actions related to resarch. Designed, created and populated tables.
Extracted data from tables using queries. Data exported to Excel for analysis.
Data in excel worksheet used for various analysis.
Table Class
Java classes
Table ClassActivity
Activities java classes
Table WorksOn
Connects subject, tasks and activities
Table Category
Categorizes activities
Table Doc
Pdf documents and web pages
Table DocActivity
Activities pdf documents and web pages
Table Library
Compiling and execution
Table LibraryActivity
Activities Compiling and execution

Figure 2: Description of the data cleaning process

We have written documentation that contains the SQL-code and the steps that must be performed to create and populate the tables. The following example describes the creation of documentation activities:

1. Create table Doc
2. Create table DocActivity
3. Insert values into Doc
4. Update fields in Doc
5. Create procedure DocActivitySituation(@sess numeric(18,0))
6. Execute the procedure for every valid session
7. Execute query for update workson_id in DocActivity
8. Execute query for update starttime and duration in DocActivity
9. Give permissions to the roles (grant select on 'tablename' to 'role')

By combining the data from the new tables with the data from the existing tables we achieved a detailed chronological overview of each participant's use of source code, compilation, execution, web pages and documents during the experiment.

The database is used by researchers with different purposes. To protect the data we found it useful to create different roles which regulated the access to the database. The roles we created are Datacleaner, ResearcherComprehension and ResearcherGuest, see Table 5. The use of roles simplifies the job of giving new users access to the data.

Table 5: Description of roles

Role	Permissions
Datacleaner	Select: All Tables, Views Create: Tables, Views, Functions, Procedures Execute: Functions, Procedures
ResearcherComprehension	Select: All Tables, Views Create: Tables Execute: Functions, Procedures
ResearcherGuest	Select: Tables ActionsClean02, Application, ClassActivity, DocActivity, LibraryActivity, Subject, Task and WorksOn

The extraction of the data was not a trivial task. By modifying previous queries we managed to extract the data we needed for the analysis. The process of data retrieval and preparation for the analysis was as follows:

1. Execute queries for ClassActivity, DocActivity and LibraryActivity for each participant and each task.
2. The results were pasted into Microsoft Excel. One spreadsheet was made for each participant.
3. Microsoft Excel's Pivot function was used to transform the data into a more comprehensible format.
4. Four charts per task were made for each participant.

Thomas, Kennedy et al. (2003) experienced in their study that the data preparation took about a full person-month. It took us one full semester each for the total work. We spent about three weeks just to comprehend the content and structure of the database. In addition we spent an amount of time understanding the procedures and queries that were crucial to our work. Much time was also spent validating the results. These challenges are described further in the next section. The tedious Microsoft Excel work described above took several weeks to complete.

Lessons Learned

There are several things we have learned. Firstly, we got a hands-on experience in the process of comprehending an unknown system written by others in order to perform maintenance tasks which is exactly what we have studied in our theses. We experienced the importance of having good documentation and source code available to understand the system before we performed the work. Secondly, we have learned the importance of documenting our work, decisions and difficulties that arose throughout the process. Thirdly, it is extremely important to verify the data. In fact, one of the greatest challenges was to get the data correct. We made several attempts before we finally succeeded. The challenges were especially related to:

- Getting all the actions when we grouped continuous actions. In addition we discovered that the summarising of duration was wrong when continuous actions were grouped.

Finally we decided to get all single actions instead of grouping those which were in order. Then we got all the actions with correct durations.

- Getting the correct window titles. When we tried to use ActionsClean02 to get the window title, we discovered that the table contained wrong window title for some actions. It was better to use the Actions table.

During the work we experienced problems we had to take care of. Firstly, we discovered that the documents had different titles depending on whether the participant had them maximized or not. This affected for instance the titles of the PDF-documents, and we had to find a solution to this problem. Another issue was that we found it necessary to categorize the activities in order to simplify the data preparation in Microsoft Excel. We decided to create another table and alter the other tables by adding a new column. Thus several updates had to be executed. If we had realized this need at a previous stage, it would have been included in the creation of the tables and perhaps saved us some work. The lesson learned is that unpredicted events will occur and must be dealt with.

Generalization and Limitations
Our contribution can be applied to similar databases and may contribute to improve the understanding of developers' program comprehension strategies. However, researches must be aware of that the SQL code written is adjusted to our experiment. If other experiments use different tasks, IDE's or applications, the code has to be modified. Even though modifications must be performed, we still point out that our contribution is of great value because of the amount of time saved by reusing our code.

3.4 Tasks

The description of the tasks is taken from Levine (2005) since this experiment is a replication of the experiment described by her. The experiment consisted of totally four different tasks. First the participants went through a training task to get familiar with the SESE-tool and experimental situation. The training task (Task 1) was a 400 LOC program (seven Java classes) which reproduced the functionality of an automated teller machine. The training task was to add a logging function to this application and was taken from Arisholm et al. (2001).

The tasks of the experiment were to modify a library application system given in Eriksson and Penker (1998). A library lends books and magazines. The books and the magazines are registered in the system. A library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are out of date or in a poor condition. The librarians can easily create, update, delete and browse information about the titles in the system. The borrowers can browse information about the titles and reserve a title if it is not available. The participants were asked to conduct the following changes on the library application system:

Table 6: Description of tasks

Task 2	Delete functionality related to ISBN number
Task 3	Extend the system to handle customers' e-mail address
Task 4	Introduce the functionality to inform a person when a loan is due

The tasks were ordered by complexity, but Task 2 and Task 3 were about the same level. Task 2 was a corrective task while task 3 and 4 were enhancement tasks.

The full task texts and detailed task description can be found in Appendix C.

3.5 The Application

The application description is also written by Levine (2005). The application consisted of four packages with totally 3600 LOC in 27 Java classes. The main class, StartClass.java, was not included in any package. This application system was used because we assumed that the application domain – a library – would be familiar to the participants.

Architectural Overview
The architecture is a traditional three-tier architecture where user interface classes access data and behavior from the persistent business objects, which store and retrieve themselves using services from the database layer. Figure 3 shows the architectural overview of the system.

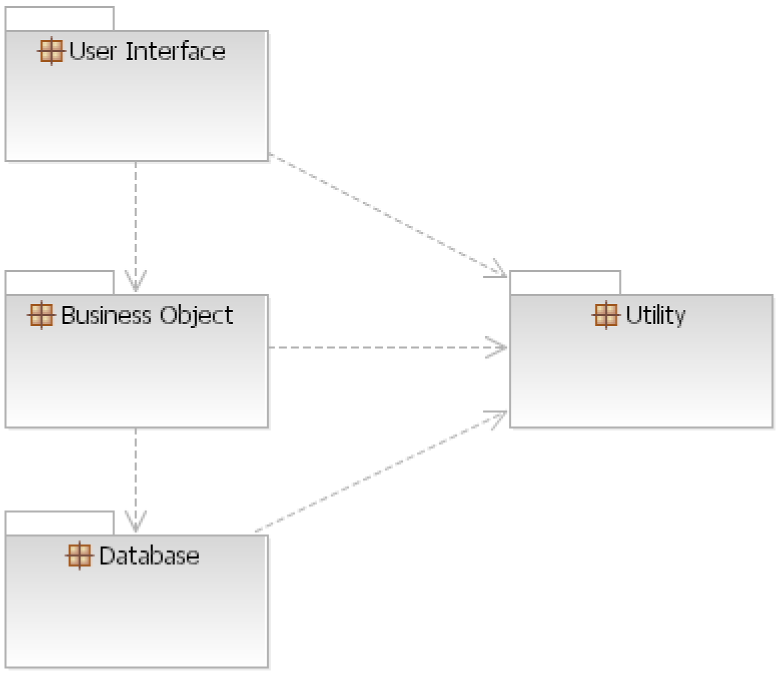


Figure 3: Architectural overview of the Unified Library Application

An overview of the classes in each package is found in Appendix D.

3.6 Documentation

The participants were provided with a document describing the Unified Library Application system (Library.pdf) and a short introduction to the Java programming language (JavaDoc.pdf). They also had access to the Java online documentation and the possibility to search for information on the internet. The Library.pdf contained an introduction to the application, detailed UML-diagrams with description of the objects and interactions and a user manual including screen dumps of the system.

The task descriptions given to the participants included test examples and screen dumps of the correct solution. The description of Task 4 also contained additional description of Java's Calendar class which was relevant for solving the task.

4 Analysis

This chapter describes the analysis of the data collected by GRUMPS and the feedback-collection, including data preparation. The data preparation for the analysis, correctness and solution time in section 4.1 are common for Tømmerberg (2006) and this thesis. The classification of developers in section 4.1.4 and the identification of comprehension strategies in section 4.2 are presented in this thesis only.

4.1 Analysis Model

This section describes the different elements used in the analysis.

4.1.1 Data Preparation for the Analysis

The data preparation for the analysis was done in collaboration with Gøril Tømmerberg. First, the data about each participant were extracted from the GRUMPS database and exported to Microsoft Excel. The queries and data are listed in Table 7. Then the data were reconciled against user actions logs produced by the GRUMPS. We identified and removed all lunch breaks or breaks longer than ten minutes from the extracted data. Thus, the variable time is the time the participants spent working excluding major non-productive breaks. After cleaning the data, Microsoft Excel's Pivot-function was used to transform the data to a comprehensible format. The different elements used in the analysis are extracted from this data. Examples of the profiles created are given in section 4.2.3.

Table 7: Description of queries to extract data from GRUMPS

Query	Data
queryClassAnalyzeMinutes	Classes visited and time spent in each class, ordered by minutes into the tasks.
queryDocAnalyzeMinutes	Documents and web pages visited and time spent in each document, ordered by minutes into the tasks.
queryLibraryAnalyzeMinutes	Start time and duration of compilation and execution of the Unified Library Application, ordered by minutes into the tasks.

Total Profile
The total profile is a graph which gives a high-level picture of the participants' actions during each task. The actions are categorized into Class, Documentation, Compile and Run Unified Library Application. Each of the categories is detailed further in the Class profile, Documentation profile and the Compile and Execution profile.

Class Profile
The class profile is a graph that presents an overview of classes visited by a participant and time spent in each class, ordered chronologically.

Documentation Profile
This is a graph that shows what kind of documents the participants have accessed during the task. The graph also shows how long the participant has been reading each document. A short description of the documents is given in Table 8.

Table 8: Description of the documents and web pages

Document	Description
Library.pdf	Describes functionality and structure of the Unified Library application. Contains UML-diagrams.
Task2.pdf	Task documentation. Includes screen dumps and test examples.
Task3.pdf	Task documentation. Includes screen dumps and test examples.
Task4.pdf	Task documentation. Includes screen dumps, test examples and information about Java's Calendar class.
JavaDoc.pdf	Short introduction to the Java programming language.
Java API	Web page. Java online documentation.
Calendar	Web page. Java class.
Google	Web page. Search for 'Java' using the Google search engine.

Compile and Execution profile
The compile and execution profile gives a detailed description of how the participants have compiled and run the Unified Library Application. It shows which windows in the application the participants have looked at and for how long.

4.1.2 Correctness

This is the assessment of the quality of the solution. Marking was on the basis of correctness and the solutions were marked as correct (1) for fully functional or not correct (0) for those with major defects. In addition the solutions were ranged from score 5 for a perfect solution to score 0 for a non-attempt.

The assessment of correctness was given by an independent consultant from another research institute. The consultant was provided with task specifications, correct solutions and guidelines for giving scores. He was not involved in the experiment or teaching the participants. All solutions were compiled, executed and thoroughly tested for functionality. The source code was also manually inspected. The total work took about 40 working hours.

4.1.3 Solution Time

This is the time in minutes to complete (understand, code and test) the change tasks. It was calculated as end_time–start_time, where start_time is the time when a participant downloaded the task description and end_time is the time when the participant uploaded his solution as recorded by the SESE tool.

These times were reconciled by Gøril Tømmerberg and me against user action logs produced by GRUMPS. Lunch breaks or other breaks longer than ten minutes were subtracted from task times. Thus, the variable time is the time participants spent on the change tasks excluding major non-productive breaks.

4.1.4 Classification of Developers

Developer Categories used in Previous Research
Different developer categories have been used in the previous research and the participants' backgrounds vary in each category. Vassdokken (2005) points out that it is difficult to categorize subjects on behalf of their expertise. A student isn't necessarily a novice, because many students nowadays have many years of work experience. On the other hand long experience does not necessary give good expertise. Hærem (2002) suggests that both demographic data, nominating and characteristic value should define a person's expertise. To be called an expert a combination of education and work experience both generally and task related should be considered together with self evaluation questions before and after the tasks.

Thomas (1998) says that within human-computer interaction the term "expert" is used rather loosely. Sometimes it refers to someone with just a few days of training, in other cases may be three of four years of practical experience. Amongst those who have studied users of several years there is a growing realisation that it takes a very long time indeed to become exceedingly good. According to Jørgensen and Sjøberg (2002) a lack of a precise definition of experience may be present in the software maintenance literature and they have found no guidelines on how to measure or interpret it. Thomas (1998) supports this by stating that there is no clear definition of what constitutes an expert user, although the term is frequently adopted in the literature.

Arisholm and Sjøberg (2004) used the categories Undergraduate and Graduate for the students and Junior, Intermediate and Senior for the professional consultants in their experiment. For the professional consultants, a project manager from each company chose consultants according to how they usually would categorize their consultants and charge for the work performed. Threats to this classification of developers are that someone who by example is considered as an intermediate consultant in one company might be considered a senior in another company.

Burkhardt et al. (1998; 2002) compared novices versus experts where the experts were professional programmers experienced in object-oriented design with C++. The mean age of the experts in Burkhardt et al. (2002) was 29.2 years and all except one were male. Their mean amount of programming experience was 9.8 years and the average amount of time for which they had used the C++ language was 3.3 years. The novices were chosen from advanced undergraduate computer science students who were experienced in C but had only basic knowledge of object-oriented programming and C++. The mean age of the novices was 26.8 years (Burkhardt et al. 2002). Seventeen were male and four were female. The average amount of student plus professional programming experience was 5.15 years. The novice participants reported an average of 0.97 years of use of C++.

Davies (2000) has also used the categories novice and expert. The group of novices were first year undergraduate computer science students, and the experts were experienced C++ programmers who were either teaching this language or used it extensively in their research. Students with different education were used in the study reported by Ko and Uttl (2003). Participants were recruited from undergraduate computer science, psychology and statistics courses. All participants had at least one introductory statistics course and experience with hypothesis testing. Jørgensen and Sjøberg (2002) studied 54, randomly selected, software maintainers in the maintenance department of a Norwegian company. Different measures were used to classify the participants on behalf of their experience.

Ye and Salvendy (1996) used ten intermediate and ten novice programmers in their experiment. The intermediate programmers were selected from graduate students who had taken at least one course in C programming. The novice programmers were selected from sophomores who had been on an introductory course in C for a half of a semester and learned the C concepts which were used in the experiment. Compared to professional programmers, the graduate students used in the experiment were not considered as experts.

Pennington et al. (1995) recruited ten professional programmers for their experiment. Three of the participants were expert procedural designers with an average of nine years of professional design and programming experience with procedural languages. Four were expert object-oriented designers with an average of 15 years of professional experience in design and programming; seven of these years were experience with object-oriented languages. The final three participants were novice object-oriented designers who were also expert procedural designers with an average of ten years of experience.

Nanja and Cook (1987) have used the three categories novice, intermediate and expert. The novices were just finishing their second term of an introductory Pascal programming course and the intermediates were finishing their third term of a junior level data structure sequence. The expert group was composed of graduate students in computer science.

To summarize, previous research vary in how many developer categories have been used. Burkhardt et al. (1998; 2002), Davies (2000) and Ye and Salvendy (1996) have used two categories whereas Pennington et al. (1995) and Nanja and Cook (1987) have used three. Arisholm and Sjøberg (2004) used five developer categories in their research. Ko and Uttl (2003) and Jørgensen and Sjøberg (2002) have not used a fixed number of categories, but have used different measures to determine the subjects' expertise.

The participants' background also differs between the studies. Pennington et al. (1995) and Jørgensen and Sjøberg (2002) have used professional developers only, whereas Arisholm and Sjøberg (2004) have used both students and professionals. Burkhardt et al.s' (1998; 2002) experts were also professionals whereas Davies' (2000) experts were experienced C++ programmers who were either teaching this language or used it extensively in their research. Both Burkhardt et al. (1998; 2002), Davies (2000) and Nanja and Cook (1987) used undergraduate computer science students for their novice category, but as Davies (2000) used first year undergraduate computer science students, Burkhardt et al.'s (2002) were advanced undergraduate computer science students. Ye and Salvendy's (1996) participants were both graduate and undergraduate students and Nanja and Cook (1987) used students that were just finishing their second term of an introductory Pascal programming course. Ko and Uttl (2003) used students with different education.

The developer grouping in the study by Arisholm and Sjøberg (2004) was done by the participating companies. Ko and Uttl (2003) participants self-reported their ability and experience and the subjects that participated in Burkhardt et al.'s (2002) and Pennington et al.'s (1995) study answered a background questionnaire. Jørgensen and Sjøberg (2002) didn't use questionnaires, but interviews because it was important to ensure interpretation of important terms. Ye and Salvendy (1996) selected their participants according to Schneiderman's (1976) classification of expertise levels. Nanja and Cook (1987) and Davies (2000) say little about the participants' background.

Developer Categories used in this Experiment
The categories used in this experiment are listed in Table 9.

Table 9: Description of developer categories

Category	Experience
Novice object-oriented	Low object-oriented programming experience
Expert procedural	High procedural programming experience Low object-oriented experience
Expert object-oriented	High object-oriented experience

In this experiment the participating companies were asked to recruit both non-experienced consultants and experienced consultants. The participants were either trainees or senior consultants in their company, and my immediate thought was to categorize the developers as novice or expert. Several related studies have used these two categories (Burkhardt et al. 1998; Davies 2000; Burkhardt et al. 2002).

To categorize the subjects I looked at the participants' resumes and the answers to the background questionnaire given in Appendix B. In their resumes I looked at their programming experience and what kind of projects they have been working on. I specially looked at their Java programming experience. In the background questionnaire I checked their self-reported programming experience (question 3.2.1) and the self-reported number of months they have been working with Java (question 4.2.1).

I found that four of the subjects have quite long programming experience, but not in the object-oriented paradigm. I first thought of categorizing them as novices, but it's likely that they have acquired lots of knowledge they could apply to the tasks in this experiment. Sharp and Griffyth (1999) found overall no evidence of negative transfer between procedural skills and the acquisition of object-oriented concepts. Rather, their results suggest a strong, positive transfer effect which outweighs any object-oriented or object-oriented-related experience and is substantially more helpful than having no development experience at all.

On the other hand, these participants were not very familiar with Java and object-oriented design. According to Pennington et al. (1995) object-oriented programming languages and design concepts may be difficult to learn. Even if there are very large benefits of reuse and a reduction of complexity with object-oriented design, these benefits can be greatly diminished if experienced personnel have major difficulties switching to the new paradigm. O'Shea[1] (1986) have found anecdotal evidence showing that people who are very experienced in procedural design methodologies have difficulty in learning object-oriented design. Thomas (1998) points out that transfer of learning usually is a positive benefit. However, negative transfer can happen, such that experience actually slows down the skill acquisition initially. In computer programming for instance, people who have very informally learned to write "spaghetti code" may actually take to a structured language slower than those with no previous programming experience.

[1] Cited by Pennington et al. (1995). I have not been able to find the original source.

Based on this knowledge I chose to use three developer categories and categorize the participants according to their object-oriented and procedural programming experience like Pennington et al. (1995) did.

Subject Information
The object-oriented experts were professional programmers experienced in object-oriented design with Java. The median age of experts was 33.5 ranges 30-38. Their median amount of programming experience was eight years ranges 5-12. The median amount of time for which they had worked with Java was four years range 2-6. They had taken from 0 to 60 credits in programming courses, the median being 30.

The object-oriented novices were also professional programmers, but had no or little programming experience neither with procedural programming languages nor with object-oriented design with Java. The median age of novices was 27.5 ranges 25-31. Their median amount of programming experience was 0.25 years ranges 0-2. The median amount of time for which they had worked with Java was 0 years ranges 0-1. They had taken from 5 to 80 credits in programming courses, the median being 35.

The procedural experts had many years of programming experience, but not much with object-oriented design and Java. Their age ranges from 32-50 the median being 38.5. The subjects have programming experience that ranges from 8-25 years the median being 12.5. But their Java work experience ranges only from 0.5-2.5 years the median being 1.5 years. They had taken from 25 to 55 credits in programming courses, the median being 35.

4.2 Identification of Comprehension Strategies

This section identifies the comprehension strategies used by the participants in this experiment. First, measures used to determine comprehension strategies in previous studies are presented followed by the definitions used in this study. Section 4.2.3 and section 4.2.4 present how the two data sources were used to identify the strategies. Finally, section 4.2.5 and section 4.2.6 show examples of the two approaches identified.

4.2.1 Measures Used in Previous Studies

In previous studies different measures have been used to determine the participants' comprehension strategies. Table 10 gives an overview of these studies.

Table 10: Overview of methods used to determine comprehension strategies in previous studies

Study	Data Collection Method	What measured	Purpose
Littman et al. (1986)	Verbal protocols	Proportion of files accessed	Breadth of comprehension
Koenemann and Robertson (1991)	Verbal protocols	Proportion of files accessed	Breadth of comprehension
Burkhardt et al. (1998)	Verbal protocols	Proportion of files accessed	Breadth and direction of comprehension
Corritore and Wiedenbeck (2000) (2001)	Screen capture software	Proportion of files accessed	Breadth and direction of comprehension
Parkin (2004)	Logging	Proportion and duration of files accessed	Breadth and direction of comprehension
Torchiano (2004)	User action capture software	Number of web pages visited and the time spent on each page	Direction of comprehension
Karahasanović et al. (2005)	User action recorder Verbal protocols	Proportion of classes accessed Use of documentation and program execution	Breadth of comprehension

In previous experiments (Koenemann and Robertson 1991; Burkhardt et al. 1998; Corritore and Wiedenbeck 2001; Parkin 2004) the numbers of files accessed have been used to identify comprehension strategies. Programmers use of abstract documentation has been interpreted by researchers as reflecting the top-down approach where the programmer try to get an overview over the application (Koenemann and Robertson 1991; Corritore and Wiedenbeck 2001; Burkhardt et al. 2002). The bottom-up approach has been characterized by the accessing of the low-level implementation files (Corritore and Wiedenbeck 2001; Burkhardt et al. 2002; Parkin 2004) or spending short time on a large number of pages (Torchiano 2004).

To identify the scope of comprehension Littman et al. (1986) used the amount of source code studied. They defined that a systematic strategy is used when the programmer reads all the code line by line. Karahasanović et al. (2005) defined the systematic strategy as reading the documentation and source code or running the application before making the required changes whereas the as-needed strategy is applied when the programmer starts to perform the maintenance tasks without first trying to understand the system.

4.2.2 Definitions

Littman et al. (1986) describe two different strategies concerning breadth of comprehension; systematic and as-needed. A programmer using the systematic strategy will typically attempt to learn how the program is constructed and how it functions before making required changes. A programmer with the as-needed strategy would only try to understand what is necessary to perform the maintenance tasks.

The identification of comprehension strategies in this study has been conducted in collaboration with Tømmerberg (2006). Our definition of the two strategies differs from other definitions which have been used. Littman et al. (1986) describe the systematic strategy as reading all the code line by line, but their program was only 250 LOC. Our application consists of 3600 LOC and is too large to be read in detail. We have therefore used the same definitions as Karahasanović et al. (2005). Table 11 gives an overview of the strategies used and what criteria we used for each category.

Table 11: Strategy definitions

Strategy	Characterized by
Systematic	Reading the documentation (Library.pdf), reading the source code or running the application before making the required changes.
As-needed	Starts to perform the maintenance tasks without first trying to understand the system.

In previous experiments (Koenemann and Robertson 1991; Burkhardt et al. 1998; Corritore and Wiedenbeck 2001) the numbers of files accessed have been used to identify comprehension strategies. However, if a participant spent only a few seconds in a class, it is not possibly to say that he was studying it. It does neither say anything about the studying of documentation or execution which can be used to understand programs.

In contrast to Karahasanović et al. (2005) we have more detailed data available. In addition to the participants' class activity profiles, we have been able to see their use of documentation and compilation and execution facilities. The data collected by the feedback-collection has been used to explain and validate the participants' actions. The next two sections describe how we have used the two sources of data in order to determine the participants' comprehension strategies.

4.2.3 Data from GRUMPS

The participants' actions were logged with timestamps in milliseconds by the GRUMPS-Lite software (Thomas et al. 2003). The data collected by GRUMPS were transformed into three levels in Microsoft Excel:
1. The raw data collected by the queries
2. Pivot tables
3. Graphs

To analyze comprehension strategies we looked at the data collected from Task 2. We began the analysis by looking at the total profile. Figure 4 shows an example of such a profile. The total profile gives of an overview of classes and documentation visited, compilation and execution performed ordered by minutes into tasks. We looked at the first third part of the total time spent during the task to determine the comprehension strategy as did Karahasanović et al. (2005). If the participant first read documentation and executed the application before spending much time in the source code, we defined it as a systematic approach. If the subject read very little documentation, did not test the program and visited several classes less than 60 seconds, the strategy was defined to be as-needed.

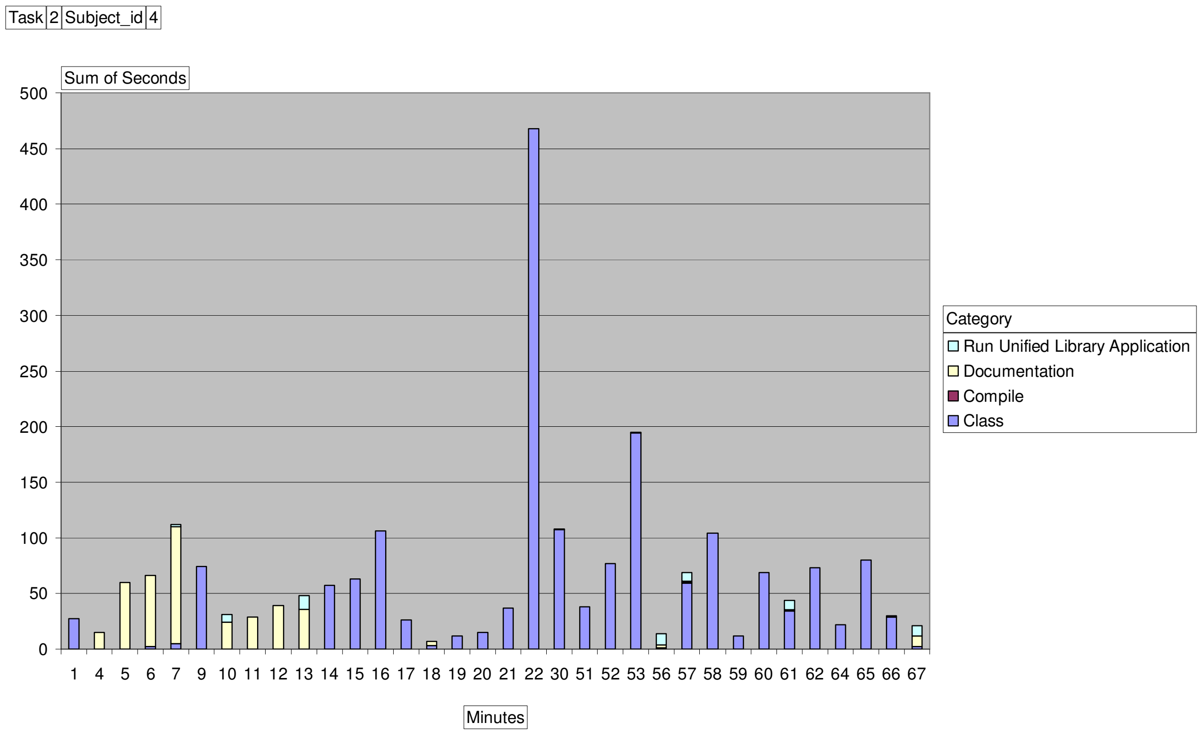


Figure 4: Example of total profile

In addition to the total profile, we looked at the detailed graphs shown in the class profile, the documentation profile and the compile and execution profile. This was necessary in order to make correct decisions by receiving detailed information on which classes they had visited, which documents they had read and how they had run the application. In some cases, the first classes visited were concerning the previous task or the .jpx-files and was thus not taken into account. For instance, in Figure 5 the first class visited was Minibank.java which belongs to the training exercise.

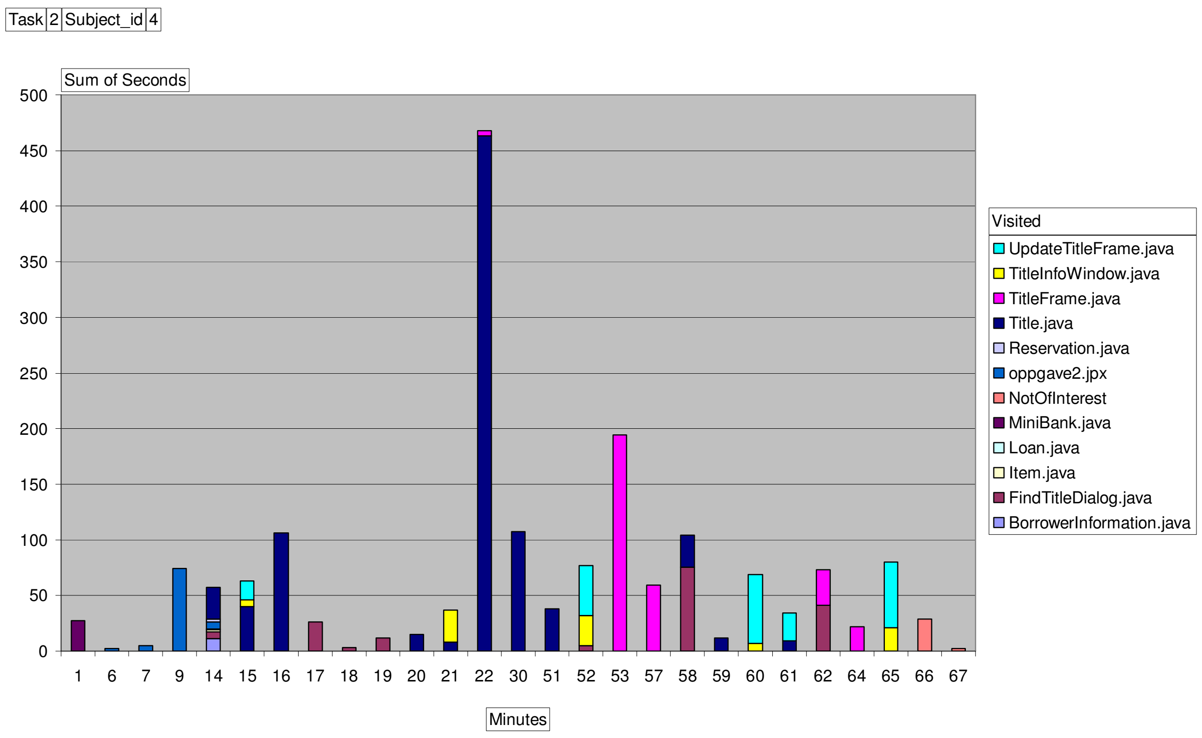


Figure 5: Example of class profile

Figure 6 shows an example of the documentation profile. We used this graph to see which documents the subject had looked at during Task 2. We specially check whether the participants had read the Library.pdf in order to determine which strategy they had used. If they had spent some time in the initial phase of Task 2 reading this document, it indicated a systematic approach.

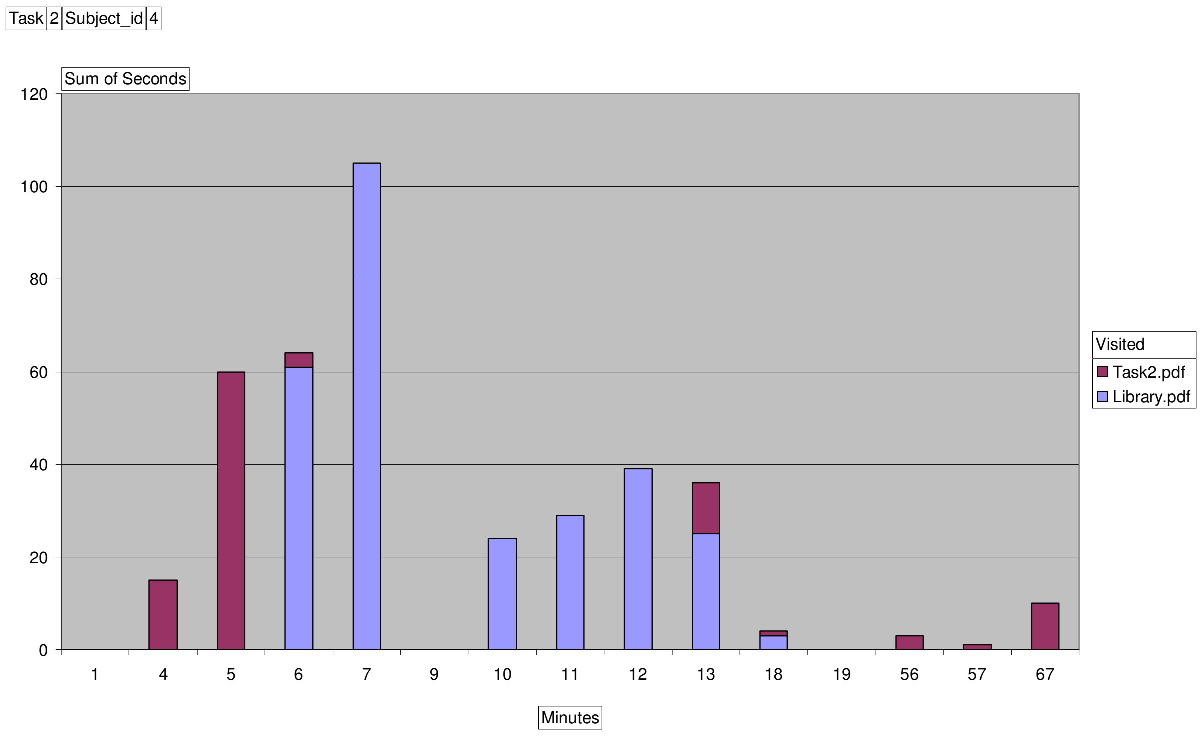


Figure 6: Example of documentation profile

The compile and execution profile gives a detailed picture of when the participants have compiled and how they have run the program. We used this profile to see if they had just opened and closed the application or whether they had tested the application thoroughly. An example of the compile and execution profile is shown in Figure 7.

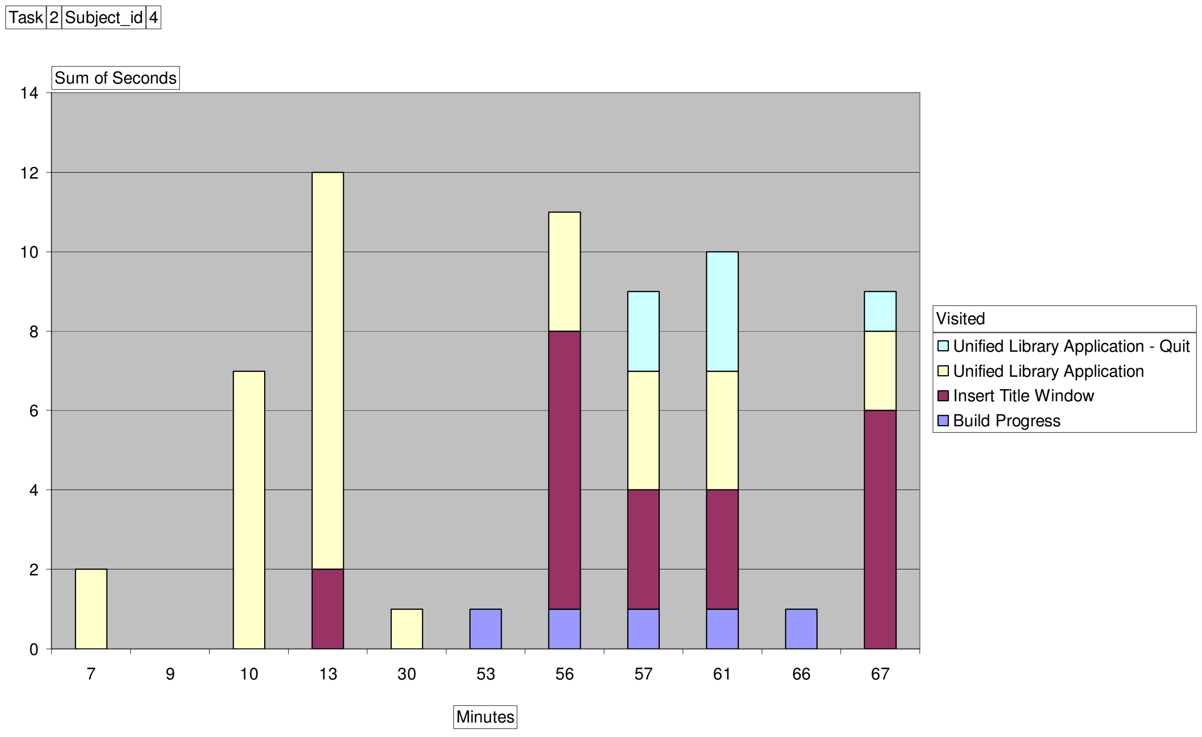


Figure 7: Example of compile and execution profile

In addition to the four graphs, we have used the raw data and the pivot tables from Microsoft Excel in our analysis. It was not always easy to see in the graphs how many classes they had visited below and over 60 seconds, how many times they had compiled and so on. Therefore we have used the raw data and the pivot tables quite extensively to count the data we needed.

So, the analysis showed two trends: either (1) the participants read the system documentation and executed the application for a longer time and visited few classes, or (2) they opened quickly many classes and spent very little or no time reading the system documentation or running the application. We assumed that the first group applied the systematic strategy because they tried to get a global overview of the application, whereas the second group applied the as-needed strategy.

4.2.4 Data from Feedback-Collection

While the data collected from GRUMPS can tell us what the participants did, it says little about why the participants chose the strategies they did. This is an advantage with the feedback-collection. Not only do they provide information about the participants' actions, but they also, to some extent, give us explanations for those actions. We have in the analysis used the data from the feedback-collection to validate our findings from the GRUMPS analysis and to help us decide the strategy when we were not sure. We were especially interested in the written feedback from Task 2 related to the participants' comprehension strategies. The comments were originally written in Norwegian and have been translated for this thesis.

These are some examples of statements from participants with the systematic strategy:

> *"I read the Task description and Library.pdf in order to get an impression of the application and how it works."*
>
> *"Get an overview of the application. Look for an UML diagram. Understand the task."*
>
> *"First, I'm planning to read the documentation and try to get an overview of the application. I assume that I also will try to run it in order to see how it works."*
>
> *"I'm currently reading the Task description. And I will execute the application and read the description to get an overview. Read source code to get an overview."*
>
> *"I have tried to get an overview of the system by reading quite thoroughly through the task and look at the class diagram and use cases. I'm thinking of using the class diagram to get an overview of where the ISBN nr is used."*
>
> *"I've begun to read the documentation and the Task description. I'm not so familiar with class diagrams, so I'm going to look a bit at that. Right now I'm planning to run the application to have a better basis when I'm reading the documentation further."*

The following statements are taken from the subjects with the as-needed strategy:

> *"I'm planning to find the value object that holds the ISBN number... Get a simple overview of the application..."*
>
> *"I have worked with removing all the ISBN references. I solved this by using the search-function in JBuilder and I commented out all lines containing "ISBN".*
>
> *"I found a place in the code where ISBN was referenced. I commented it out, got errors from the compiler, commented more until I got no errors. Tested the application and saw that the ISBN still was visible in the user interface. Went to the UI-code and removed all the ISBN. I didn't bother thinking of what was really going on in the code. I just went on removing the code and hoped it would work. And so it did."*
>
> *"I'm thinking that I have to search for all lines containing ISBN and comment them out. After that run the application and see if it works."*

4.2.5 Example of Systematic Approach

This section gives an example of a typical systematic approach by showing the Task 2 profiles for subject 12.

The total profile shown in Figure 8 is an example of the systematic strategy. During the first third part of Task 2 the participant mainly read documentation and tested the application to get an overview of the system. After the first third part he edited classes and tested the modifications.

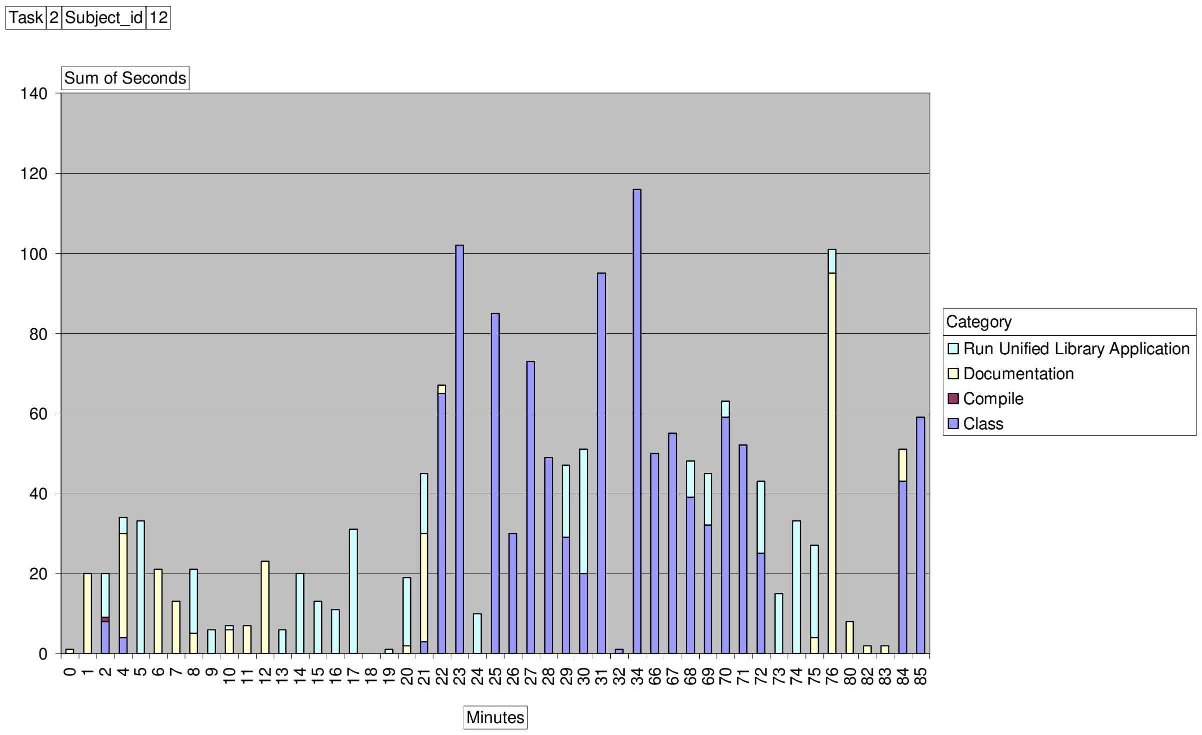


Figure 8: Example of total profile – Systematic approach

The documentation profile for subject 12 is shown in Figure 9. The graph illustrates that the participant first read the task description and then the system documentation in order to understand the program.

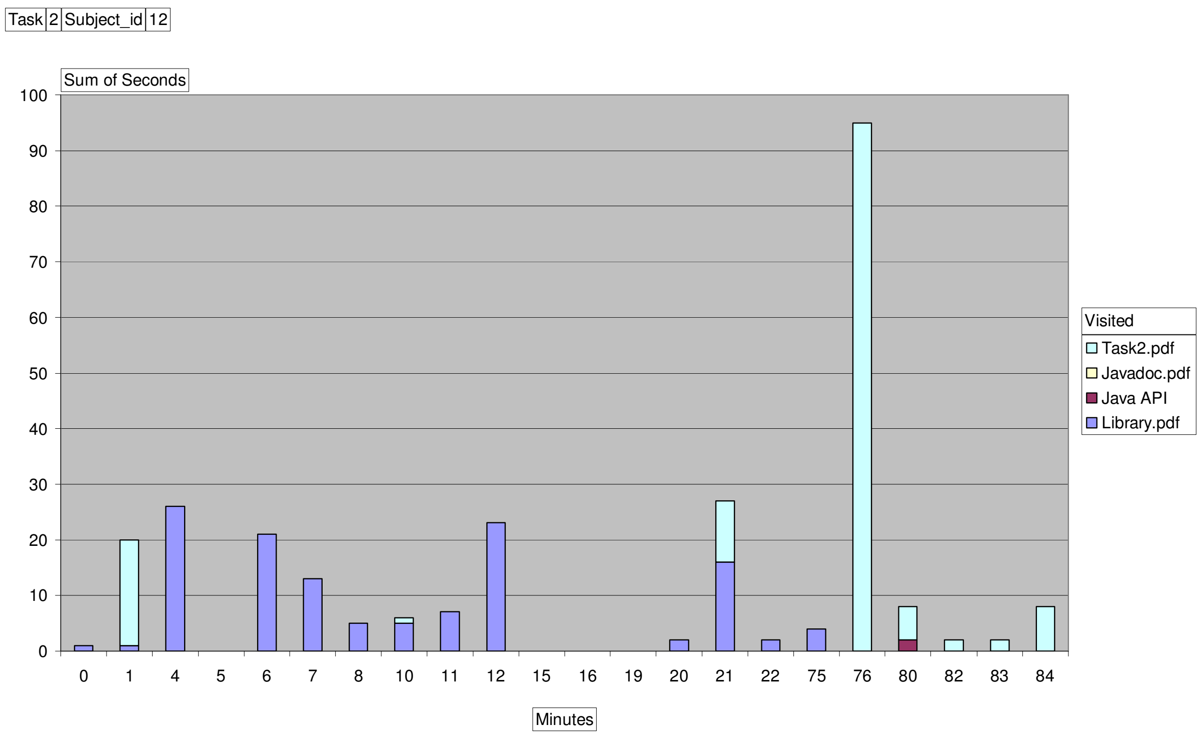


Figure 9: Example of documentation profile – Systematic approach

The class profile for subject 12 in Figure 10 illustrates that the participant didn't go the classes until 22 minutes had passed. This participant had a clear plan where to make the changes before looking at the code and spent most of the time in the classes that needed to be edited in order to perform the task.

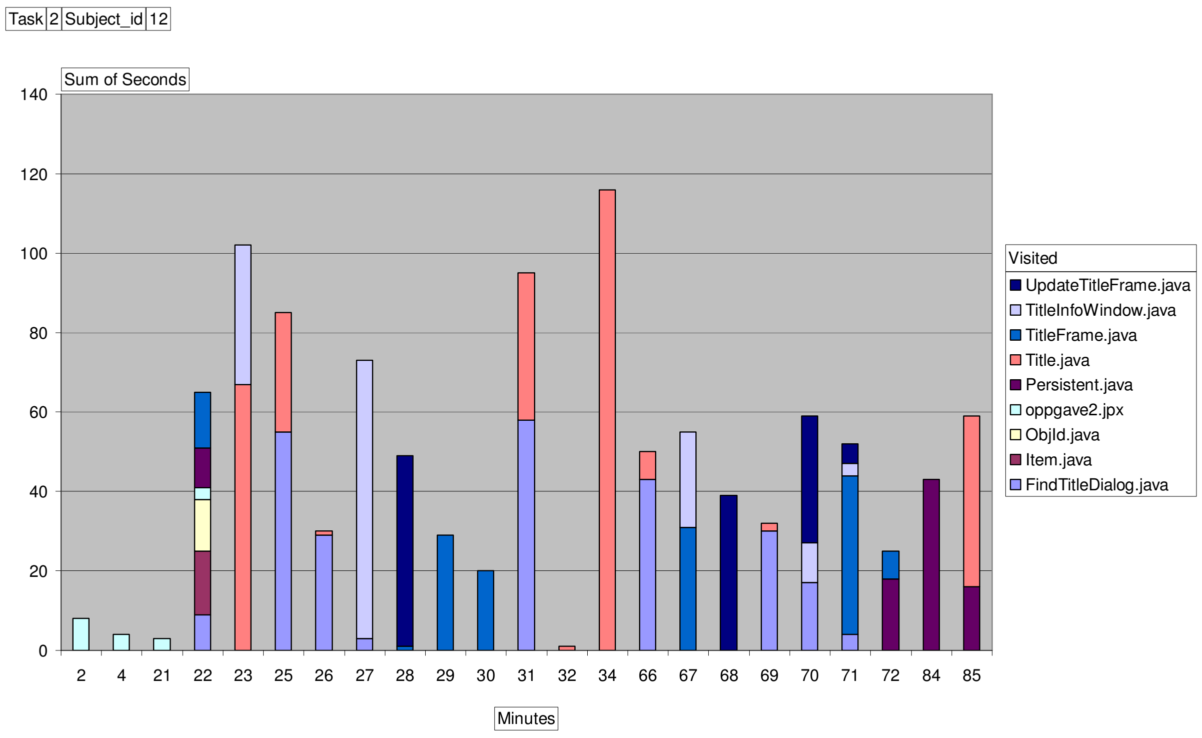


Figure 10: Example of class profile – Systematic approach

The compile and execution profile in Figure 11 shows that the participants tested several parts of the application before he went to the source code. This gave him insight in the dynamic aspects of the system.

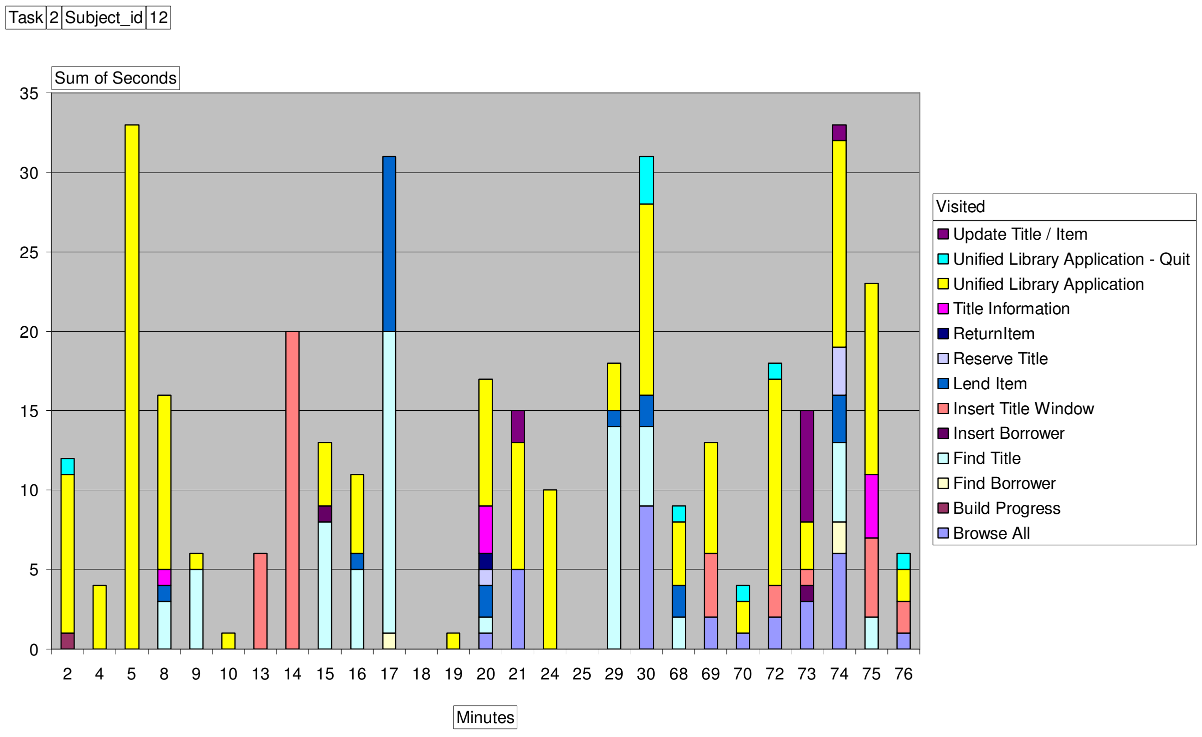


Figure 11: Example of compile and execution profile – Systematic approach

Our observations from the different profiles were confirmed by the result from the feedback-collection:

> *"I'm trying to understand the application. I have registered some books and a borrower. Now I'm reading the description of the system, but not as thorough as I first planned. I'm mainly getting an overview. I want to start at the task pretty fast, but want to get an overview first."*

This confirmed that the developer had used a systematic comprehension approach.

4.2.6 Example of As-needed Approach

This section gives an example of a typical as-needed approach by showing the Task 2 profiles for subject 16.

Figure 12 shows the total profile for subject 16 who used the as-needed strategy. He almost went straight to the code without spending much time reading documentation or testing the application to get an overview.

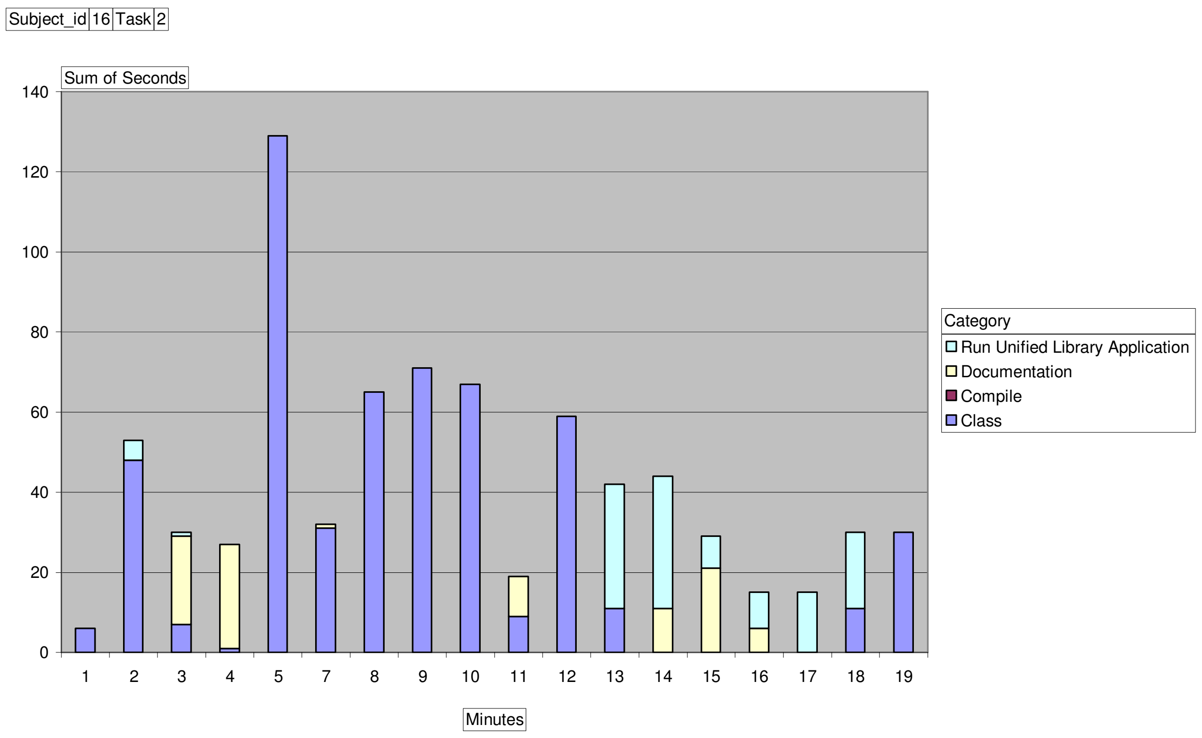


Figure 12: Example of total profile – As-needed approach

The graph in Figure 13 shows that the subject did not try to get an overview of the system by first reading the system documentation. He only read the task description in the first third part of the task solving and did not look at the Library.pdf until the end of the task.

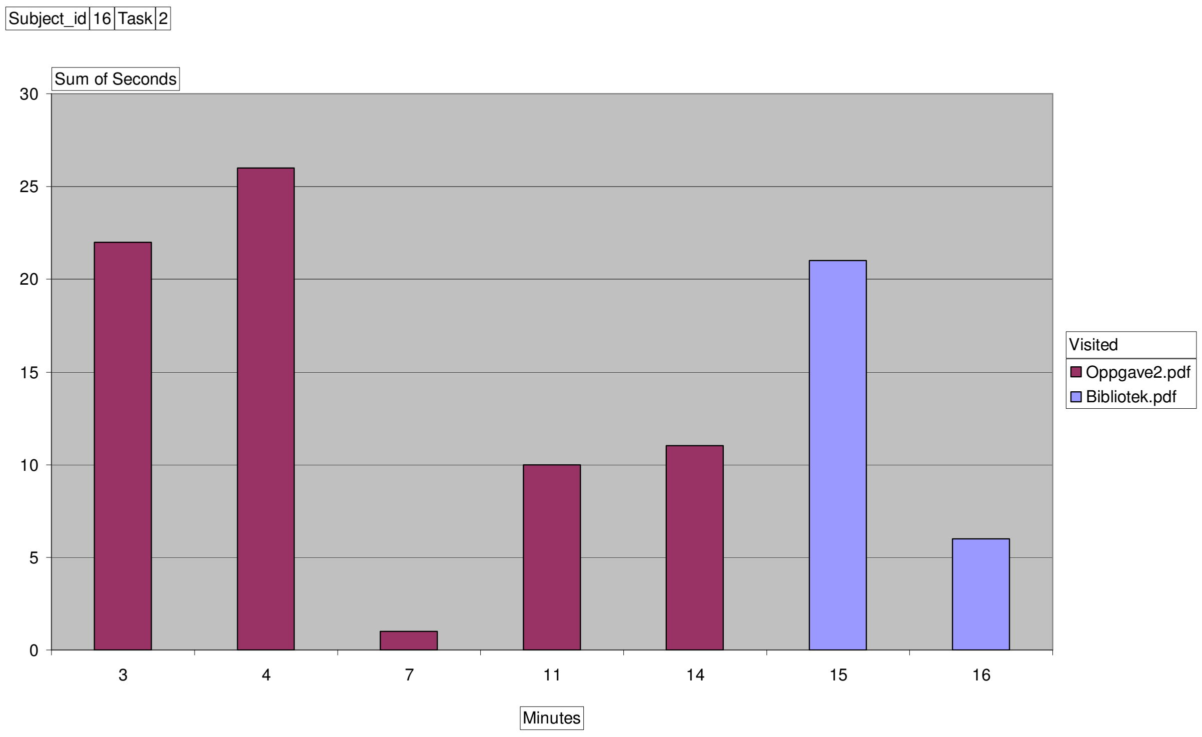


Figure 13: Example of documentation profile – As-needed approach

The class profile for subject 16 in Figure 14 shows that the subject visited several classes during the first third of the task and only one more than 60 seconds. He also visited almost all the classes of the program, not only those to be altered in the task.

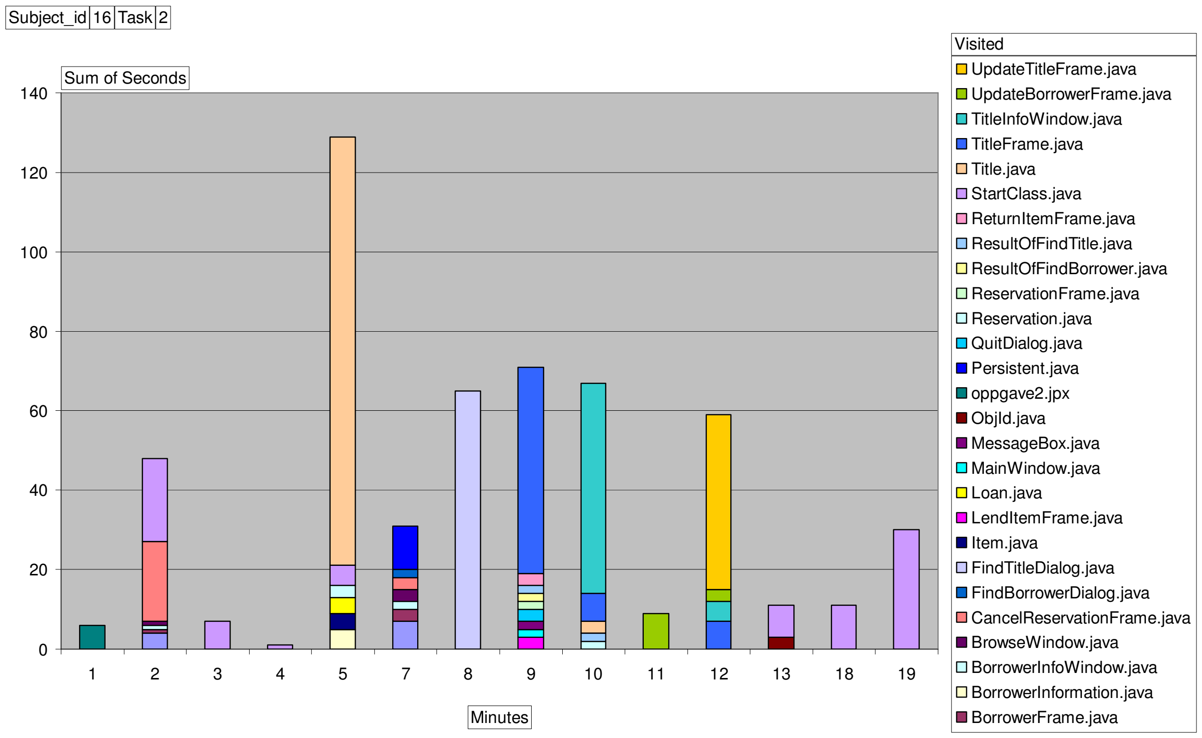


Figure 14: Example of class profile – As-needed approach

Figure 15 illustrates how the participant ran the Unified Library Application. As one can see from the chart, he only opened and closed the program to see that it worked in the beginning of the task. He did not test any functionality until 13 minutes had passed.

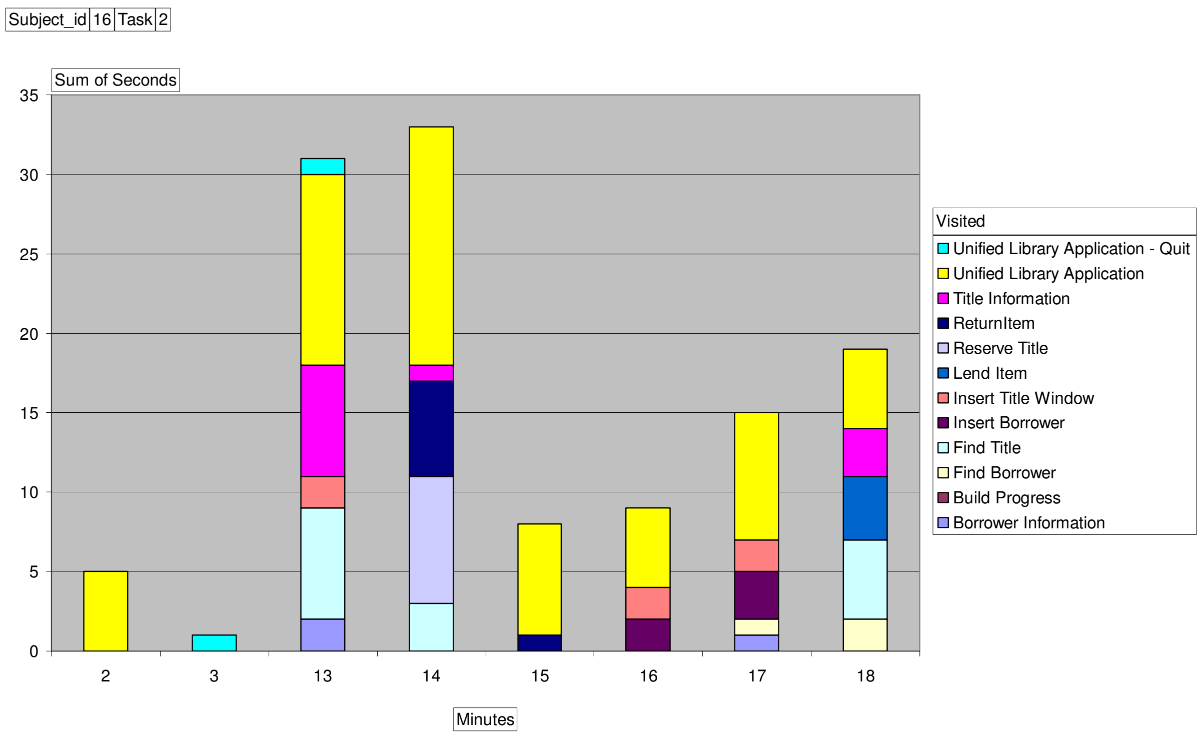


Figure 15: Example of compile and execution profile – As-needed approach

In the data from the feedback-collection we found no comments that indicated that he had tried to get an overview of the system before he performed the maintenance tasks. This made us confident that the participant had used an as-needed approach.

> *"I have worked with removing all the ISBN references. I solved this by using the search-function in JBuilder and I commented out all lines containing "ISBN".*

5 Results

This chapter presents the results of the study. The objective of this research was to identify the comprehension strategies of professional developers maintaining an object-oriented application. Furthermore we wanted to identify the effects of strategies on performance and the effects of expertise on strategies and performance. Section 5.1 presents the results regarding comprehension strategies. Section 5.2 presents the results on effects of strategies on performance whereas section 5.3 presents the results on effect of expertise on strategies and performance. Finally, section 5.4 summarizes the results.

5.1 Comprehension strategies

Littman et al. (1986) describe two different strategies concerning breadth of comprehension; systematic and as-needed where the systematic strategy is defined as reading all the code line by line. For the purpose of this analysis, we have described the systematic approach to be an approach where the participant reads the documentation or runs the application in order to get an overview before performing the maintenance tasks. The as-needed approach is defined as an approach where the participants goes straight to the code to perform the maintenance tasks without first trying to understand the system.

Based on the analysis of the data collected by GRUMPS and the feedback-collection method, we found that 18 participants used the systematic strategy and six participants the as-needed strategy (Table 12). Table with all the participants and their strategies is given in Appendix A.

Table 12: Results from strategy analysis

Strategy	Number of participants	Percentage
As-needed	6	25%
Systematic	18	75%
Total	24	100%

The results of the strategy analysis differ somewhat from what have been found in other studies. Littman et al. (1986) found that the professional programmers that maintained a 250 LOC Fortran program either used a systematic or an as-needed strategy. Half of the participants used each strategy. Koenemann and Robertson (1991) found that none of the twelve expert programmers in their experiment followed a truly systematic strategy of comprehension when modifying a 636 LOC Pascal program. The subjects were only interested in the parts relevant to the modification task and spent a major part of their time searching for code segments and no time understanding parts of the program. Koenemann and Robertson attribute their differences from Littman et al. to the fact that their program was larger in size. However Mayrhauser and Vans (1994) observed an instance of systematic study of a large program.

Our application was 3600 LOC and too large to be read line by line. If we had defined the systematic strategy to be reading all the code lines, none of the participants would probably have used the systematic strategy. However, we found that the participants executed the application and read the system documentation in order to understand the program. In Littman et al.'s (1986) and Koenemann and Robertson's (1991) experiments, the subjects were not allowed to test and debug their enhancements.

The research presented by Levine (2005) identifies the comprehension strategies used by novice programmers. A controlled experiment with 39 students was conducted with the same maintenance tasks and application used in this study. Levine found that according to the results from the GRUMPS database 18 participants used as-needed strategy and 20 participants used the systematic strategy. According to the results from the verbal protocols, twelve participants used as-needed strategy and 17 participants used systematic strategy. The total number of participants was higher in the GRUMPS analysis because they did not collect verbal protocols for all the participants. Levine compared the results for each of the participants from both the GRUMPS analysis and the verbal analysis and put those whose strategies didn't match in the unknown category. Her findings are listed in Table 13.

Table 13: Strategy distribution from study by Levine (2005)

Strategy type	Number of participants
As-needed	12
Systematic	15
Unknown	11
Total	38

The results in Table 13 imply that 56% of the participants applied the systematic strategy and 44% the as-needed strategy of those whose strategy was known. It is my opinion that the results in this study differ from Levine's because the participants in her experiment were students. Research has shown that experts often use a systematic approach to the program comprehension opposite to novices who tend to use more as-needed approach (Nanja and Cook 1987).

5.2 Effects of Strategies on Program Comprehension

This section presents the results found by comparing the performance of the participants applying the two strategies with respect to the correctness of solution, use of compilation and execution facilities, use of documentation and the solution time.

5.2.1 Strategy vs. Correctness

We have compared the results of the participants from the different strategies in regard to the correctness of the solutions. The results are shown in Table 14.

Table 14: Number of correct solutions for each strategy type

Strategy type	Number of participants	Number of correct solutions Task 2	Number of correct solutions Task 3	Number of correct solutions Task 4
As-needed	6	0	5	3
Systematic	18	10	15	8
Total	24	10	20	11

None of the subjects that used an as-needed approach produced a correct solution for Task 2. On the other hand, 55.6% of those who used a systematic strategy solved Task 2 correctly. This shows that the group that applied the systematic strategy performed significantly better than those who used the as-needed strategy. This finding is consistent with Levine (2005) who studied students performing the same tasks on the same application as used in this study. Levine also found that the overall quality of the solutions was better for the participants with a systematic strategy in Task 2. Research done by Littman et al. (1986) confirms that there is a strong relationship between using a systematic approach to acquire knowledge about the program and modifying the program successfully.

Task 2 was the easiest task, but only ten of the 24 participants managed to produce a correct solution. The data from the feedback-collection shows that the participants might have underestimated the complexity of the task:

> *"I have just finished Task 2. It seems to have gone well commenting out the code by using the search-function in JBuilder". [Subject 7]*
>
> *"The last task (Task 2) was very easy and I hope this one is too." [Subject 21]*

The assessment of the solutions for these two participants shows that they both had errors in their solution. The most common errors for all the participants were error in removal of else and one label declaration. One participant also reported that he was not very experienced in GUI-programming. Thus, Task 2 was not as trivial as they thought.

The results for Task 3 were the same regardless of which strategy the participants used that is 88.3% of the participants produced a correct solution, respectively. Levine (2005) found in her research that the participants that applied the systematic strategy also performed better for Task 3. However, she also found that the difference in average quality differed less for Task 3 than for Task 2.

Task 4 was the last and the most difficult task. Table 14 shows that the subjects who used the as-needed approach performed slightly better than those who used a systematic strategy. 50% of the participants using the as-needed approach produced a correct solution opposite to 44.4% of the systematic participants. Corritore and Wiedenbeck (2000), who studied the scope of comprehension-related activities for procedural and object-oriented programmers found that the programmers over time built a broad view of the program, regardless of strategy. They argue that even though the programmer might not try to comprehend more of the program than necessary in order to solve the task at first, knowledge will accumulate over different tasks on the same program until a broad scope of comprehension is achieved. However, one explanation for the results of Task 4 might be due to the fact that the participants had to use the Calendar class. The success of the task solving depended on their knowledge regarding this more than their knowledge of the application. Several of the subject reported that they experienced problems using this class and that they had not used it much before.

5.2.2 Strategy vs. Compilation and Execution

This section compares the results of the participants from the different strategies in regard to the use of compilation and execution facilities. Table 15 shows the relation between the strategies and the number of times the participants compiled the application.

Table 15: Compilation during the tasks vs. strategy

Strategy type	Number of participants	Number of times compiled Task 2			Number of times compiled Task 3			Number of times compiled Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
As-needed	6	4	18	7	15	72	25.5	6	47	17.5
Systematic	18	2	24	11	5	48	18.5	0	35	16.5
Total	24	2	24	9.5	5	72	19.5	0	47	16.5

For Task 2 the median number of times compiled was eleven for the participants that employed the systematic strategy and seven for the participants that employed the as-needed strategy. Data from the feedback-collection shows that the participants used the compiler to find instances of ISBN that had to be removed:

> *I use the compiler to find ISBN in the GUI. I remove objects and see where I get syntax errors. [Subject 6]*

> *I first looked in the class diagram to find out where the ISBN field is found. Thereby I opened the class to remove the field and the method that uses the field. In the tool I normally use, I would have used the "find usage"-function. In JBuilder I use the "make project"-function to see where the system fails instead. [Subject 11]*

> *I have started up the work by removing references to ISBN in the source code. First I commented out references in the Title class. Then I tried to compile the code, and the error message showed some other references that I also removed. [Subject 12]*

In Task 3 the participants with an as-needed approach compiled significantly more than those participants that used a systematic approach (median 25.5 and 18.5). The minimum and maximum values are also much higher for the as-needed group. Table 15 also shows that all the participants compiled more during Task 3 regardless of which strategy they chose. Task 3 was more complex than Task 2 and consisted of three parts. Thus it required more compilation than in Task 2. The results for Task 4 were reverse that is the median was higher for the as-needed group than the systematic group (median 17.5 and 16.5). The difference is though little.

Table 16 shows the relation between the strategies and the percentage of the total solution time the participants spent executing the Unified Library Application. The percentages are calculated by dividing the total time executing the application with the solution time. This was done in order to be able to compare the participants.

Table 16: Program execution during the tasks vs. strategy

Strategy type	Number of participants	Execution (% of solution time) Task 2			Execution (% of solution time) Task 3			Execution (% of solution time) Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
As-needed	6	4.3	11.7	8.05	5.3	12.2	9.55	3.8	7.9	4.4
Systematic	18	2.1	17.4	9.75	2.7	25.6	8.6	0	14.6	6.3
Total	24	2.1	17.4	9.2	2.7	25.6	8.85	0	14.6	6.05

The median for the participants that applied the systematic strategy is 9.75 and 8.05 for the participants that applied the as-needed strategy. It is not surprising that the median is higher for those who chose a systematic strategy as our definition of the systematic approach involved program execution. Those who used the as-needed approach spent more time executing the program in Task 3 than the participants that used the systematic approach (median 9.55 and 8.6 respectively). For Task 4 the medians are 6.3 (systematic) and 4.4 (as-needed).

Littman et al. (1986) suggest that programmers using the as-needed strategy may depend more heavily upon testing and debugging to learn about the program's structure than programmers using the systematic strategy. But they also mention that finding and correcting errors in the design of an enhancement before implementing the enhancement is most efficient. Programmers using the systematic strategy may be able to avoid or detect errors in the design of an enhancement more efficiently than programmers using the as-needed strategy. Programmers using the as-needed strategy may have to depend upon testing and debugging to correct errors in the design of the enhancement after the enhancement has been implemented. An excerpt from the feedback-collection data illustrates the importance of testing the program before modifying it.

> *I'm now starting on the next task. I'm reading the task description and executing the application. In the previous task I forgot to first run the program. When I experienced errors later on, I didn't know whether they were old errors or new errors introduced by me. [Subject 10]*

The results from this experiment is not supported by Littman et al.'s (1986) suggestions. The participants using an as-needed strategy did not test the program significantly more than those who used a systematic strategy except from Task 3.

5.2.3 Strategy vs. Documentation

Table 17 compares the results of the participants from the different strategies in regard to the percentage of the total solution time the participants spent reading documentation. The documentation includes all PDF-files and web pages as described in Table 8, section 4.1.1. The percentages are calculated by dividing the total time reading documentation with the solution time. This was done in order to be able to compare the participants.

Table 17: Use of documentation during the tasks vs. strategy

Strategy type	Number of participants	Documentation (% of solution time) Task 2			Documentation (% of solution time) Task 3			Documentation (% of solution time) Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
As-needed	6	4.2	14.4	5.8	3.7	11.4	5.8	3.6	15.2	9.2
Systematic	18	0.8	58.5	9.95	1.5	15.5	5.25	0	15.5	8.55
Total	24	0.8	58.5	8.8	1.5	15.5	5.55	0	15.5	8.75

The subjects that used a systematic approach read obviously more documentation doing Task 2 than those who used an as-needed approach (median 9.95 and 5.8). This is due to our definition of the two strategies where the systematic strategy involved reading the system documentation. But for the two last tasks, the participants using the as-needed strategy spent more time reading documentation. The differences between the two groups are though very little.

In the study by Littman et al. (1986) documentation was mainly seen as the last resort and only consulted when other methods of comprehension failed. Many subjects reported that they had bad experiences with useless documentation. Littman et al. say that subjects try to avoid the extra effort of studying documentation if they believe that the information can be obtained directly from the code. On the other hand, subjects do use documentation when they know that code itself does not provide the desired information. In this experiment the participants had to use the Calendar class in Task 4 and several participants reported that they not were familiar with this class. Accordingly, they had to read documentation to solve the task. The median for the participants that applied the as-needed strategy is higher in Task 4 than in Task 2 and Task 3. Also the participants that used the systematic strategy spent more time reading documentation in Task 4 than in Task 3.

Singer et al. (1997) found that software engineers reported that reading documentation was the activity they performed the most. However, when following a software engineer and a whole group of software engineers in their daily work, they found interestingly that reading documentation was a relatively infrequent activity for both the group and the individual software engineer.

In this research we found that the programmers read the documentation both in the beginning of the task solving to get an overview of the system and during the three tasks. In contrast to Littman et al. (1986) and Singer et al. (1997) reading documentation was not an infrequent activity, but counted for about 5-10% of the total solution time.

5.2.4 Strategy vs. Solution Time

Table 18 shows the results of the participants from the different strategies in regard to the total solution time.

Table 18: Task solution time vs. strategy

Strategy type	Number of participants	Solution Time (minutes) Task 2			Solution Time (minutes) Task 3			Solution Time (minutes) Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
As-needed	6	15	40	28.5	38	101	59.5	49	91	67
Systematic	18	19	60	34	38	86	53.5	0	120	57
Total	24	15	60	34	38	101	54.5	0	120	57.5

Table 18 shows that the median solution time increases with the complexity of the tasks. The participants who used the systematic strategy spent more time performing Task 2. The median solution time for the participants that used the systematic approach was 34 minutes opposite to 28.5 minutes for the participants that used the as-needed approach. For Task 3 and Task 4 the results were reversed that is the participants that applied the as-needed approach spent more time solving the tasks.

One explanation for these results can be that during Task 2, the participants with the systematic strategy spent a fair time getting to know the program before solving the maintenance tasks. They achieved an overview of the system, but this is time consuming. The participants that applied the as-needed approach, however, did not spend much time on getting to know the system. Instead they spent all their time solving the task. When it was time for Task 3, the participants who had used the systematic strategy already had knowledge about the program beyond what they learned from solving Task 2. This allowed them to spend less time on Task 3. For the participants with the as-needed strategy, who didn't have this broad knowledge, Task 3 was more time consuming.

Task 4 was the most complex task and required knowledge about the Calendar class. One of the participants that used the as-needed approach reported that he had to learn about the Calendar class and that this could be time consuming.

Young (1996) claim that the systematic approach requires more time during the comprehension stage of maintenance but results in a strong cognitive model of the program. The as-needed approach requires less comprehension time but will result in a weaker and incomplete cognitive model. In the research described by Levine (2005) with novices, the participants with a systematic approach spent more time on Task 2 than the participants with an as-needed approach, but for Task 3 the situation was reversed. This corresponds to the findings in this experiment with professional developers. Thus, the results of this research confirm and extend previous studies and today's knowledge of program comprehension.

5.3 Effects of Expertise on Program Comprehension

We have categorized the participants into three developer categories based on their expertise. The number of participants in each category is listed in Table 19. Table with all the participants and their expertise can be found in Appendix A.

Table 19: Category distribution of developers

Category	Number of participants
Object-oriented novice	8
Procedural expert	4
Object-oriented expert	12
Total	24

The following sections present the results of the expertise's effect on comprehension strategies, the correctness of solution, use of compilation and execution facilities, use of documentation and the solution time.

5.3.1 Expertise vs. Strategy

The relation between strategy applied and developer category is listed in Table 20.

Table 20: Expertise vs. strategy

Developer category/ Strategy	Number of participants	As-needed	As-needed percentage	Systematic	Systematic percentage
Object-oriented novice	8	4	50%	4	50%
Procedural expert	4	1	25%	3	75%
Object-oriented expert	12	1	8.3%	11	91.7%
Total	24	6	25%	18	75%

Table 20 shows that it is a clear difference in the approaches between the object-oriented novices, the procedural experts and the object-oriented experts. Amongst the six as-needed approaches found, four of the participants were object-oriented novices. This means that 50% of the object-oriented novices used an as-needed approach opposite to only 8.3% of the object-oriented experts. 25% of the procedural experts also used the as-needed strategy.

Nanja and Cook (1987) have analyzed the on-line debugging process of novices, intermediates and expert student programmers. They found that experts first attempt to gain a high level understanding of the program and how it functions while novices use a bottom-up approach to understand the program. The debugging strategy used was the major difference between the experts and the novices and intermediates. The experts employed a comprehension approach in which they first attempted to understand the program and then used this knowledge to locate and correct errors. Intermediates and novices did not try to understand the system before debugging. Vessey (1985; 1986) investigated the verbal protocols of eight novice and eight expert programmers for program debugging. He also found that experts used the top-down breadth-first strategy to understand the program before debugging the program, whereas novices directly started the search for program bugs. Ye and Salvendy (1996) found the use of an overall top-down strategy by both intermediates and novices for program comprehension. Novices' control strategies involved however more opportunistic elements than experts' in the overall top-down process of program comprehension.

Burkhardt et al. (1998) found that overall subjects were similar in the scope of their comprehension, although the experts tended to consult more files. They found strong evidence of top-down, as well as multiple guidance in expert comprehension. They also found less use of top-down processes in novice comprehension.

Gilmore (1990) points out that one of the distinguishing features of expert programmers is the fact that they possess a repertoire of strategies, and are capable of choosing an appropriate strategy based on the programming situation, tasks and language requirements. Davies (1993) has also found in his study that experts appear to have a greater range of strategies available to them which leads to greater flexibility in performance.

The study of professional programmer has shown that the as-needed approach was applied most by the object-oriented novices. This is supported by previous research thus our results confirm and extend today's knowledge of effect of expertise on the breadth of comprehension.

5.3.2 Expertise vs. Correctness

We have compared the results of the participants from the different categories in regard to the correctness of the solutions. The results are shown in Table 21.

Table 21: Number of correct solution for each developer category

Developer category	Number of participants	Number of correct solutions Task 2	Number of correct solutions Task 3	Number of correct solutions Task 4
Object-oriented novice	8	2	7	1
Procedural expert	4	1	4	3
Object-oriented expert	12	8	9	7
Total	24	11	20	11

The object-oriented experts performed better on Task 2 than the two other groups of participants. 66.7% of the object-oriented experts managed to produce a correct solution opposite to only 25% of the procedural experts and 25% of the object-oriented novices. I believe that these results are related to the strategies the participants applied. The object-oriented novices and the procedural experts used the as-needed approach more than the object-oriented experts and did not gain a broad understanding of the system before they performed the maintenance task like the object-oriented experts did. Previous studies have shown that there is a correlation between using the systematic strategy and modifying a program successfully (Littman et al. 1986; Karahasanovic et al. 2005).

The results for Task 3 are quite different. Here 87.5% of the object-oriented novices managed solve the task correctly. All of the four expert procedural subjects produced a correct solution, while the object-oriented experts showed the weakest results (75% correct solutions).

The fourth and final task shows diverse results between developer categories. Here the object-oriented novices performed extremely badly. Only one of the eight participants managed to solve the task correctly. This might be due to their little experience with the Calendar class. Several of the participants reported that they had problems regarding this. The procedural experts performed quite well with a 75% correctness score. On the other hand, only 58.3% of the object-oriented experts produced correct solutions.

Nanja and Cook (1987) found in their analysis of the online debugging process that experts were the most successful in correcting all of the errors. They state that studies have shown that experts make fewer errors and locate and correct bugs faster than novices. Novices frequently add additional bugs during debugging while experts rarely do. All the experts in their study successfully located and corrected all six bugs, two of the intermediates failed on one logical error, and only two novices found all the errors. Our findings correspond to their result as the object-oriented experts in this study performed averagely well for all tasks and the procedural experts did very well on Task 2 and Task 3. On the other hand, the object-oriented novices had a low correctness percentage both in Task 2 and Task 4. However, Jørgensen and Sjøberg (2002) found that the most experienced maintainers did not predict maintenance problems better than maintainers with little or medium experience

5.3.3 Expertise vs. Compilation and Execution

Table 22 shows the relation between the developer categories and the number of times the participants compiled the application.

Table 22: Compilation during the tasks vs. expertise

Developer category	Number of participants	Number of times compiled Task 2			Number of times compiled Task 3			Number of times compiled Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
Object-oriented novice	8	4	23	17.5	18	72	27.5	6	47	19.5
Procedural expert	4	3	9	5.5	14	19	17	2	17	16
Object-oriented expert	12	2	24	11	5	46	16	0	31	15
Total	24	2	24	9.5	5	72	19.5	0	47	16.5

The object-oriented novices compiled significantly more than the two other groups of programmers. Table 22 shows that the minimum and maximum values and the median are higher during all the three tasks for the novices. The two experts groups show quite the same compilation behaviour both in Task 3 and 4, but the object-oriented experts compiled more than the procedural experts in Task 2 (median 11 and 5.5).

Jadud (2005) has observed novice compilation behaviour as it naturally occurred in classroom tutorial sessions. He found that 51% of all compilation events occurred less than 30 seconds after the previous event and points out that it is true that novices recompile often, which confirms the results of this experiment.

Table 23 shows the relation between the developer categories and the percentage of the total solution time the participants spent executing the Unified Library Application. The percentages are calculated by dividing the total time executing the application with the solution time. This was done in order to be able to compare the participants.

Table 23: Program execution during the tasks vs. expertise

Developer category	Number of participants	Execution (% of solution time) Task 2			Execution (% of solution time) Task 3			Execution (% of solution time) Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
Object-oriented novice	8	5.4	17.4	9.65	3.8	12.2	9.3	3.8	9.2	6.05
Procedural expert	4	4.3	16.3	8.05	8.5	9.6	8.85	5.5	14.6	10.3
Object-oriented expert	12	2.1	14.4	9.75	2.7	25.6	8.1	0	10.9	5.5
Total	24	2.1	17.4	9.2	2.7	25.6	8.9	0	14.6	6.05

The object-oriented experts spent less time executing the application throughout the tasks. The results are opposite for the procedural experts who ran the program more during the tasks. The object-oriented novices show the same trend as the object-oriented experts.

Nanja and Cook (1987) found in their analysis of the online debugging process that novices made three times as many and intermediates twice as many program runs as experts. The novices also hand simulated the execution of one or more of the procedures in the program making notes about the values of the variables at various stages of the program. No expert explicitly hand executed the program. There are small differences between the three groups of participants in this experiment. They spent about the same percentage of the solution time executing the application in all the tasks except from Task 4 where the procedural experts ran the program the most. Thus, there was no significant effect of expertise in the participants' execution behaviour.

5.3.4 Expertise vs. Documentation

Table 24 shows the relation between the strategies and the percentage of the total solution time the participants spent reading documentation. The percentages are calculated by dividing the total time reading documentation with the solution time. This was done in order to be able to compare the participants.

Table 24: Use of documentation during the tasks vs. expertise

Developer category	Number of participants	Documentation (% of solution time) Task 2			Documentation (% of solution time) Task 3			Documentation (% of solution time) Task 4		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
Object-oriented novice	8	5	21.9	10.8	3.7	15.5	7.75	3.6	15.2	8.75
Procedural expert	4	0.8	9	3.05	3	6.3	5.5	6.6	10.6	10.45
Object-oriented expert	12	5.2	58.5	9.2	1.5	9.2	5.15	0	11.2	7.15
Total	24	0.8	58.5	8.8	1.5	15.5	5.6	0	15.2	7.9

The results in Table 24 show that the object-oriented novices spent 10.8% of the solution time reading documentation in Task 2. The object-oriented experts also spent a fair amount of reading documentation (median 9.2) in Task 2. The median for the procedural experts is on the other hand only 3.05.

The object-oriented novices also spent more time reading documentation in Task 3 than the object-oriented experts and the procedural experts (median 7.75 opposite to 5.15 and 5.5). The object-oriented experts did not spend that much time reading the documentation as the other particpants in Task 4. This can be due to their longer expericence with the Calendar class. One interesting observation is that procedural experts read very little documentation in the first task (median 3.05), but the median increased in Task 3 (median 5.5) and Task 4 (median 10.45).

5.3.5 Expertise vs. Solution Time

Table 25 shows the results of the participants with different expertise in regard to the total solution time.

Table 25: Task solution time vs. expertise

Developer category	Number of participants	Solution Time (minutes) Task 2			Solution Time (minutes) Task 3			Solution Time Task 4 (minutes)		
		Min	Max	Median	Min	Max	Median	Min	Max	Median
Object-oriented novice	8	15	50	26.5	38	101	57	46	91	65.5
Procedural expert	4	27	41	35	41	84	63	28	67	57
Object-oriented expert	12	19	60	36.5	38	86	49.5	0	120	57
Total	24	15	60	34	38	101	54.5	0	120	57.5

The object-oriented novices spent less time than the procedural and object-oriented experts solving Task 2 (median 26.5, 35 and 36.5). I address this finding to the fact that novices used the as-needed strategy more than the experts. The systematic strategy is more time consuming than the as-needed approach but results in a strong cognitive model of the program.

The object-oriented experts solved Task 3 faster (median 49.5) than the object-oriented novices (median 57) and the procedural experts (median 63). In Task 4 the median was 57 minutes both for the object-oriented and the procedural experts while the median was 65.5 for the novices. Nanja and Cook (1987) found that experts were the fastest in correcting the errors which is also found in this experiment in Task 3 and Task 4 when comparing the object-oriented novices and experts. However, Ye and Salvendy (1996) experienced that the effect of expertise on the performance time was not significant.

5.4 Summary

In this research the comprehension strategies of professional programmers maintaining a 3600 LOC Java application have been identified. Two different approaches to the task solving were found. In the first one, called systematic strategy, the participants tried to get an overview of the application before performing the maintenance tasks, by reading the system documentation or running the application. 75% of the participants applied this strategy. In the second one, called as-needed strategy, the participants went straight to the task solving, without trying to acquire any knowledge of the system. This strategy was applied by 25% of the participants. This result differs from previous studies in that no one has found that so many of the participants applied the systematic strategy.

5.4.1 Effects of Strategies on Program Comprehension

The performance of the participants applying the two strategies have been compared with regard to the correctness of solution, use of compilation and execution facilities, use of documentation and the solution time. Task 2 was the easiest task, but only ten of the 24 participants managed to produce a correct solution. This might be due to their difficulties with GUI-programming and that they might have underestimated the complexity of the task. The participants that applied the systematic strategy performed better than those who applied the as-needed strategy. Research confirms that there is a strong relationship between using a systematic approach to acquire knowledge about the program and modifying the program successfully (Littman et al. 1986; Karahasanovic et al. 2005). The participants that used the systematic strategy used more time performing Task 2, but less time solving Task 3 and Task 4. One explanation for these results can be that during Task 2, the participants employing the systematic strategy spent a fair time getting to know the program before solving the maintenance tasks. This allowed them to spend less time on Task 3 and Task 4.

The results for Task 3 were the same regardless of which strategy the participants applied. The subjects who used the as-needed approach performed slightly better than those who used the systematic strategy in the final task. This might indicate that the participants that used the as-needed approach gained a broader view of the program over time which is supported by Corritore and Wiedenbeck (2000). However, another explanation for the results of Task 4 might be due to the fact that the participants had to use the Calendar class and the success of the task solving depended on their knowledge regarding this more than their knowledge of the Unified Library Application.

Furthermore I have looked at how much time of the task solving the participants spent reading documentation, executing the application and the number of times they compiled. As the definition of the systematic strategy involved program execution and reading documentation, the participants that applied the systematic approach spent more time on these activities than the participants using the as-needed approach in Task 2. Littman et al. (1986) suggest that programmers using the as-needed strategy may depend more heavily upon testing and debugging to learn about the program's structure than programmers using the systematic strategy. The results from this experiment are not supported by Littman et al.'s suggestions. The participants using an as-needed strategy did not test the program significantly more than those who used a systematic strategy except from in Task 3. However, the participants with an as-needed approach compiled the application a lot more than those participants that used a systematic approach in Task 3. They also spent more time reading documentation in the two last tasks than the participants that used the systematic strategy.

5.4.2 Effects of Expertise on Program Comprehension

The participants in this study were 24 professional programmers with different programming experience. Eight participants were object-oriented novice, four participants were procedural expert and twelve participants were object-oriented expert. The three groups of participants have been compared with regards of comprehension strategy, correctness of solution, use of compilation and execution facilities, use of documentation and the solution time.

It was a clear difference in the comprehension approaches between the three groups. The object-oriented novices applied a more as-needed approach than the two other groups of participants. Similar results have been found in previous research (Vessey 1985; Vessey 1986; Nanja and Cook 1987). The object-oriented novices also had a low correctness percentage both in Task 2 and Task 4. Nanja and Cook (1987) found in their analysis of the online debugging process that experts were the most successful in correcting all of the errors. They state that studies have shown that experts make fewer errors and locate and correct bugs faster than novices. However, Jørgensen and Sjøberg (2002) found that the most experienced maintainers did not predict maintenance problems better than maintainers with little or medium experience

The object-oriented novices compiled significantly more than the two other groups of participants. The minimum and maximum values and the median for number of times they compiled were higher during all the three tasks for the novices. That novices recompile often is also found in similar studies (Jadud 2005). There were no significant effects of expertise in the participants' execution behaviour except from that the object-oriented experts spent less time executing the application throughout the tasks while the procedural experts and novices did not. The results are not consistent with previous studies that showed that novices made three times as many program runs as experts (Nanja and Cook 1987).

6 Validity

The most important threats to the validity of the experiment are discussed in this chapter. Some of the threats mentioned can also be found in Tømmerberg (2006), but there they are specially related to her research.

6.1 Experimental Design

In this experiment we wanted to log all user actions in order to determine the participants' comprehension strategies and see if the strategies and expertise had any effects on program comprehension. All available material was presented online only. Thus the participants didn't receive any documentation in hard-copy. We wanted to make the experimental environment as close to the participants' normal work environment as possible, but the restrictions on use of hard-copy documentation may have caused limitation regarding that. Some participants might prefer reading on paper over reading electronically and this can have affected their behaviour. However, none of the participants expressed the lack of hard-copies or books in the feedback-collections.

One of the participants expressed that the situation not was totally realistic in that they knew they not were going to work with the program and tasks after the experiment had finished. If they knew that they eventually had to solve the tasks correctly like in a real work situation, they probably would have put more effort into the work.

6.2 Participants

All the participants were professional developers who worked in five different software companies. The companies selected themselves the participants for this experiment. The differences in skills among developers are considerable, and we don't know whether those selected were representative for the companies or if they sent their least valuable employees. It is also possible that the companies recruited their best consultants in order to achieve a good reputation. The results indicate that it was a mixture of highly skilled and novices who participated in the experiment.

6.3 Program and Tasks

The library application used in the experiment was 3600 LOC, and can be considered to be a medium-size application, according to the classification given by von Mayrhauser and Vans (1994). The application and tasks were larger than those typically used in software engineering experiment. However, the programs and tasks were smaller than real-world programs and tasks, as reported by the observational study of von Mayrhauser and Vans (1997). It is possible that the results would be different for larger applications and more complex tasks. Our results are thus limited to situations in which the programmers had to comprehend and maintain a medium system, previously unknown to them and developed by others.

6.4 Analysis

The data collected by GRUMPS has been used extensively in this study and this section discusses several threats to validity of the analysis of the data collected.

The process of data cleaning and preparation was difficult. Even though we carefully verified all data, there may be activities which have been left out. We excluded activities which were not related to solving the tasks e.g. reading online newspapers. Breaks below ten minutes have not been removed and might have given an incorrect picture of the actions. We can not say whether they had a break or actually looked at the active window. In our analysis we have assumed the latter.

The graphs' scales of the different participants varied depending on the time spent on the various activities. If a participant spent a short time on all activities, they can appear as longer than they really are compared to scales were the participant spent longer time on some of the activities. It was therefore very important to look at the scales during the analysis of the graphs, but we might have been misled by the scales in our analysis.

The participants' use of classes was used to determine the participants' comprehension strategies. However, we can't see whether they edited or read the code. A separation of the reading and editing would make the results more accurate regarding the identification of comprehension strategies. If we saw that a participant spent time in the initial phase reading source code, we'd say he had a systematic approach. On the other side, if the participant used his time in the classes editing, we would define his approach to be as-needed.

The identification of the participants' comprehension strategies has been quite subjective except from the analysis of class activity where we looked at how many classes they had visited in the first third of Task 2. We also checked how many of these classes they had looked at less than and more than one minute. However, we have not had the same precise measures regarding documentation and execution activity and thus the strategies were sometimes difficult to identify.

7 Conclusions and Future Work

Software maintenance is widely recognised as an important part of a programmer's work. A central activity in software maintenance is program comprehension which is the process of understanding program code unfamiliar to the programmer. An important issue in program understanding is the effect of expertise on comprehension. The objective of this research was to identify the comprehension strategies of professional developers maintaining an object-oriented application. Furthermore we wanted to examine the effects of strategies on performance and the effects of expertise on strategies and performance. To address these issues, we conducted a controlled experiment with 24 professional programmers. They had different levels of expertise and were introduced to an unknown 3600 LOC Java application on which they performed three maintenance tasks.

There are numerous studies both on program comprehension (Koenemann and Robertson 1991; Corritore and Wiedenbeck 2000; Corritore and Wiedenbeck 2001; Burkhardt et al. 2002) and on effect of expertise on program comprehension (Nanja and Cook 1987; Burkhardt et al. 1998; Burkhardt et al. 2002). This experiment has been conducted in a more realistic environment by using a larger program and by providing compilation and execution facilities. We found that the participants used program execution to understand the application before they went to the tasks. This questions the validity of previous studies that lacked these facilities (Koenemann and Robertson 1991; Parkin 2004). Karahasanović et al. (2005) have conducted an experiment with the same application and tasks as used in this study, but in contrast to their study our participants were professional developers. We also had better control of the subjects' actions and by developing a data analysis tool we achieved a more detailed picture of the participants' actions and strategies. In addition, written feedback was collected and used to explain and validate the participants' actions.

7.1 Contributions

This section presents the contributions of the research and suggests implications based on the results.

7.1.1 The Further Development of GRUMPS

The participants' actions were logged during the experiment by means of the GRUMPS software. Data preparation for the analysis of low-level data is an extremely difficult process because often unwieldy and unmanageable data is generated. One of the contributions of this research is the further development of GRUMPS. We have cleaned and extracted low-level data to get detailed information of each participant's actions during the tasks. This was not a trivial task. We experienced several difficulties during the work, but managed to retrieve the data we needed for the analysis. The extracted data gave a very detailed picture of the participants' use of documentation, compilation and execution in addition to source code. The documentation and SQL-code written can be reused by researchers in similar studies and contribute to improve the knowledge of developers' comprehension strategies.

7.1.2 Identification of Professional Developers' Comprehension Strategies

In this research the comprehension strategies of professional programmers maintaining a 3600 LOC Java application have been identified. We found that the participants used two different strategies to solve the tasks. One group read the system documentation and executed the program in order to get an overview of the application before performing the maintenance tasks. This was identified as a systematic approach. The second group went straight to the task solving without trying to acquire any knowledge of the system. Hence, they didn't know much about the system on which the tasks should be performed. This approach was identified as as-needed.

Twenty-four programmers participated in the experiment, and 18 of these used the systematic approach. The as-needed approach was applied by six participants. This result differs from other studies in that no one has found that so many of the participants applied the systematic strategy. Previous research has found that half of the participants applied each of the strategies (Littman et al. 1986) and that none of the participants applied a truly systematic strategy (Koenemann and Robertson 1991). Karahasanović et al. (2005) found that 56% of the participants applied the systematic strategy while 44% of the participants used an as-needed approach. I address the differences in the strategy distribution to the fact that their participants were students while the participants in this study were professional programmers. The identification on professional programmers' comprehension strategies contribute to extend today's knowledge of program comprehension.

7.1.3 Effects of Strategies on Program Comprehension

The performance of the participants applying the two strategies have been compared with regard to the correctness of solution, use of compilation and execution facilities, use of documentation and the solution time. Task 2 was the easiest task, but only ten of the 24 participants managed to produce a correct solution. This might be due to their difficulties with GUI-programming and that they might have underestimated the complexity of the task. None of the subjects that used the as-needed strategy produced a correct solution for Task 2. On the other hand, 55.6% of those who used the systematic strategy solved Task 2 correctly. Research confirms that there is a strong relationship between using a systematic approach to acquire knowledge about the program and modifying the program successfully (Littman et al. 1986; Karahasanovic et al. 2005). The participants that used the systematic strategy spent longer time performing Task 2, but less time solving Task 3 and Task 4. One explanation for these results can be that during Task 2, the participants employing the systematic strategy spent a fair time getting to know the program before solving the maintenance tasks. This allowed them to spend less time on Task 3 and Task 4.

The results for Task 3 were the same regardless which strategy the participants applied. The subjects who used the as-needed approach performed slightly better than those who used a systematic strategy in the final task. One explanation for the results of Task 4 might be due to the fact that the participants had to use the Calendar class and the success of the task solving depended on their knowledge regarding this more than their knowledge of the Unified Library Application.

Furthermore I have looked at how much time of the task solving the participants spent reading documentation and executing the application. As the definition of the systematic strategy involved program execution and reading documentation, the participants that applied the systematic approach spent more time on these activities than the participants using the as-needed approach in Task 2. Littman et al. (1986) suggest that programmers using the as-needed strategy may depend more heavily upon testing and debugging to learn about the program's structure than programmers using the systematic strategy. The results from this experiment are not supported by Littman et al.'s suggestions. The participants using an as-needed strategy did not test the program significantly more than those who used a systematic strategy except from in Task 3. However, the participants with an as-needed approach compiled the application a lot more than those participants that used a systematic approach in Task 3. They also spent more time reading documentation in the two last tasks than the participants that used the systematic strategy.

7.1.4 Effects of Expertise on Program Comprehension

The participants in the experiment had three levels of expertise. One group had little object-oriented experience and were thus grouped as novice object-oriented. The second group had long programming experience, but with the procedural paradigm mostly. These participants were called expert procedural. The rest of the participants had worked quite long with object-oriented programming and were categorized as expert object-oriented. From the total number of 24 participants in the experiment, we found that eight participants were object-oriented novice, four participants were procedural expert and twelve participants were object-oriented expert.

When comparing the different levels of expertise with the two comprehension strategies, we found that it was a clear difference in the approaches between the three groups. 50% of the object-oriented novices used an as-needed approach opposite to only 8.3% of the object-oriented experts and 25% of the procedural experts. Similar results have been found in previous research (Vessey 1985; Vessey 1986; Nanja and Cook 1987) and this research confirm and extend today's knowledge of effect of expertise on program comprehension strategies.

The object-oriented novices had a low correctness percentage both in Task 2 and Task 4. Nanja and Cook (1987) found in their analysis of the online debugging process that experts were the most successful in correcting all of the errors. They state that studies have shown that experts make fewer errors and locate and correct bugs faster than novices. However, Jørgensen and Sjøberg (2002) found that the most experienced maintainers did not predict maintenance problems better than maintainers with little or medium experience

The object-oriented novices compiled significantly more than the two other groups of participants. The minimum and maximum values and the median for number of times they compiled were higher during all the three tasks for the novices. That novices recompile often is also found in similar studies (Jadud 2005). There were no significant effects of expertise in the participants' execution behaviour except from that the object-oriented experts spent less time executing the application throughout the tasks while the procedural experts and novices did not. The results are not consistent with previous studies that showed that novices made three times as many program runs as experts (Nanja and Cook 1987).

7.1.5 Implications

The research presented in this thesis has several implications. Firstly, two different approaches to program comprehension have been identified. Programmers should be aware of that these two different two strategies exist and know when to use them successfully. Secondly, this research has found that the object-oriented and procedural experts maintained the system more successfully than the object-oriented novices. Research has found that experts appear to have a greater range of strategies available to them which leads to greater flexibility in performance. Thus, novices should learn from the experts in how to apply the strategies in order to maintain a system successfully. They should gain insight into how professional programmers use documentation, how frequently they compile code and test partial solutions. One way of attaining this goal is to expose novices to how an expert programmer works as done in this study.

7.2 Future Work

In further research several issues can be addressed. In this experiment all participants' actions were logged and stored in the GRUMPS database. We have made a data analysis tool and have got a detailed picture of the participants' most relevant actions. Since the material in the GRUMPS database is quite extensive, it can be exploited further. For instance which windows in the Unified Library Application the participants have looked at can be investigated in more detail. This would give a richer picture of how the participants got to know and tested the application.

In future studies more precise measures of the strategies should be applied. One could apply quantitative measures for instance that a participant must have read documentation for more than one minute, tested the application longer than one minute etc. to use the systematic strategy. This would simplify the identification of comprehension strategies and make the decisions less subjective.

A more detailed study of how the subjects have used the system documentation would be very useful. The Library.pdf document could be split into several documents. This would give a more precise picture of which part of the documentation the participants have used to comprehend the program e.g. the class diagram, sequence diagram, screen dumps, written text etc. This will be addressed in further studies.

Bibliography

Arisholm, E. and D. I. K. Sjøberg (2004). "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software." IEEE Trans. Softw. Eng. **30**(8): 521-534.

Arisholm, E., et al. (2002). "A web-based support environment for software engineering experiments." Nordic J. of Computing **9**(3): 231-247.

Arisholm, E., et al. (2001). "Assessing the Changeability of two Object-Oriented Design Alternatives--a Controlled Experiment." Empirical Software Engineering **6**(3): 231.

Bennedsen, J. and M. E. Caspersen (2004). Teaching Object-Oriented Programming: Towards Teachning a Systematic Programming Process. 8th European Conferance on Object-Oriented Programming, Oslo, Norway.

Biggerstaff, T. J., et al. (1994). "Program understanding and the concept assignment problem." Commun. ACM **37**(5): 72-82.

Blinman, S. and A. Cockburn (2005). Program comprehension: investigating the effects of naming style and documentation. Proceedings of the Sixth Australasian conference on User interface - Volume 40. Newcastle, Australia, Australian Computer Society, Inc.

Brooks, R. (1983). "Towards a theory of the comprehension of computer programs." Int. J. Man-Mach. Stud. **Vol. 18, no. 6**: 543-554.

Burkhardt, J.-M., et al. (1998). The effect of object-oriented programming expertise in several dimensions of comprehension strategies. IWPC '98. Proceedings., 6th International Workshop on Program Comprehension.

Burkhardt, J.-M., et al. (2002). "Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase." Empirical Softw. Engg. **7**(2): 115-156.

Busvold, J. (2006). Problem and problem solving strategies in maintenance of object-oriented systems [In preparation]. Department of Informatics. Oslo, University of Oslo. **Master of Science**.

Corritore, C. L. and S. Wiedenbeck (2000). Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study. Proceedings of the 8th International Workshop on Program Comprehension, IEEE Computer Society.

Corritore, C. L. and S. Wiedenbeck (2001). "An exploratory study of program comprehension strategies of procedural and object-oriented programmers." Int. J. Hum.-Comput. Stud. **54**(1): 1-23.

Davies, S. P. (1993). "Models and theories of programming strategy." Int. J. Man-Mach. Stud. **39**(2): 237-267.

Davies, S. P. (2000). Expertise and the comprehension of object-oriented programs. 12th Workshop of the Psychology of Programming Interest Group, Cozenza, Italy.

Eriksson, H.-E. and M. Penker (1998). Case Study. UML Toolkit. New York, John Wiley & Sons, Inc.

Evans, H., et al. (2003). "The pervasiveness of evolution in GRUMPS software." Softw. Pract. Exper. **33**(2): 99-120.

Fitzgerald, S., et al. (2005). Strategies that students use to trace code: an analysis based in grounded theory. Proceedings of the 2005 international workshop on Computing education research. Seattle, WA, USA, ACM Press.

Gilmore, D. J. (1990). Expert programming knowledge: A strategic approach. Psychology of Programming, Computers and People Series. T. R. G. Green, R. Samurcay and D. J. Gilmore. London, Academic Press**:** Chapter 3.2. 223-234.

Hærem, T. (2002). "Task Complexity and Expertise as Determinants of Task Perception and Performance: Why Technology-Structure Research has been unreliable and inconclusive." Series of Dissertations.

InterMedia. (2006). "InterMedia COOL homepage." Retrieved 11.04, 2006, from http://www.intermedia.uio.no//projects/research/cool_en.html;jsessionid=54B4ECF0CD3277506D3078E7429D2973.

Jadud, M. C. (2005). "A First Look at Novice Compilation Behaviour Using BlueJ." Computer Science Education **Vol. 15, No. 1, March 2005**: 25 – 40.

Jones, C. T. (1998). Estimating Software Costs. New York NY, McGraw-Hill.

Jørgensen, M. and D. I. K. Sjøberg (2002). "Impact of experience on maintenance skills." Journal of Software Maintenance **14**(2): 123-146.

Karahasanovic, A. (2005). Comprehension of Object-Oriented Systems, Simula Research Laboratory.

Karahasanovic, A., et al. (2005). "Comprehension strategies and difficulties in maintaining object-oriented systems: an explorative study." The Journal of Systems and Software.

Ko, A. J., et al. (2005). Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA, ACM Press.

Ko, A. J. and B. Uttl (2003). Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems. Proceedings of the 11th IEEE International Workshop on Program Comprehension, IEEE Computer Society.

Koenemann, J. and S. Robertson (1991). Expert problem solving strategies for program comprehension. Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology. New Orleans, Louisiana, United States, ACM Press.

Levine, A. K. (2005). A Study of Comprehension Strategies and Difficulties by Novice Programmers Performing Maintenance Tasks on Object-Oriented Systems. Department of Informatics, University of Oslo. **Master of Science**.

Littman, D. C., et al. (1986). Mental models and software maintenance. Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers. Washington, D.C., United States, Ablex Publishing Corp.

Misanchuk, E. R. and R. Schwier (1992). "Representing interactive multimedia and hypermedia audit trails." Journal of Educational Multimedia and Hypermedia **1**(3): 355-372.

Nanja, M. and C. R. Cook (1987). An analysis of the on-line debugging process. Empirical studies of programmers: second workshop, Ablex Publishing Corp.**:** 172-184.

O'Brien, M. P., et al. (2004). "Expectation-based, inference-based, and bottom-up software comprehension." **16**(6): 427-447.

O'Shea, T. (1986). "Panel: The learnability of object-oriented programming systems." Object-oriented programming systems, languages and applications: Proceedings of OOPSLA '86: 502-504.

Parkin, P. (2004). An Exploratory Study of Code and Document Interactions during Task-directed Program Comprehension. Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), IEEE Computer Society.

Pennington, N. (1987). Comprehension strategies in programming. Empirical studies of programmers: second workshop, Ablex Publishing Corp.**:** 100-113.

Pennington, N., et al. (1995). "Cognitive Activities and Levels of Abstraction in Procedural and Object- Oriented Design." Human-Computer Interaction **Volume 10**: 171-226.

Ramalingam, V. and S. Wiedenbeck (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. Papers presented at the seventh workshop on Empirical studies of programmers. Alexandria, Virginia, United States, ACM Press.

Reeves, T. C. and J. G. Hedberg (2003). Interactive Learning Systems Evaluation.

Renaud, K. and P. Gray (2004). Making sense of low-level usage data to understand user activities. Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. Stellenbosch, Western Cape, South Africa, South African Institute for Computer Scientists and Information Technologists.

Rugaber, S. (1995). Program Comprehension, Georgia Institute of Technology.

Schneiderman, B. (1976). "Exploratory experiments in programmer behaviour." International Journal of Computer and Information Sciences **5**: 123-143.

Shaft, T. M. (1995). "Helping programmers understand computer programs: the use of metacognition." SIGMIS Database **26**(4): 25-46.

Sharp, H. and J. Griffyth (1999). "The Effect of Previous Software Development Experience on Understanding the Object-Oriented Paradigm." Journal of Computers in Mathematics and Science Teaching **18(3)**: 245-265.

Singer, J., et al. (1997). An examination of software engineering work practices. Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research. Toronto, Ontario, Canada, IBM Press.

Storey, M. A. D., et al. (2000). "How do program understanding tools affect how programmers understand programs?" Science Of Computer Programming **36**(2-3): 183-207.

Thomas, R. C. (1998). Long term human-compter interaction: an exploratory perspective, Springer-Verlag London.

Thomas, R. C., et al. (2003). Generic usage monitoring of programming students. Interact, Integrate, Impact: Proceedings of the 20th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education, Adelaide, 7-10 December 2003.

Thomas, R. C. and R. Mancy (2004). Use of large databases for group projects at the nexus of teaching and research. Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education. Leeds, United Kingdom, ACM Press.

Torchiano, M. (2004). Empirical investigation of a non-intrusive approach to study comprehension cognitive models. Proceedings of the Eight Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), IEEE Computer Societey.

Tømmerberg, G. (2006). Comprehension-Related Activities during Maintenance of Object-Oriented Systems: An In-Depth Study. Department of Informatics. Oslo, University of Oslo. **Master of Science**.

Vassdokken, R. (2005). The background information on subjects in program comprehension studies. Department of Informatics. Oslo, University of Oslo. **Master of Science**.

Vessey, I. (1985). "Expertise in debugging computer programs: A process analysis." INT. J. MAN-MACH. STUD. **23**(5): 459-494.

Vessey, I. (1986). "Expertise in debugging computer programs: an analysis of the content of verbal protocols." IEEE Trans. Syst. Man Cybern. **16**(5): 621-637.

Visaggio, G. (1999). "Assessing the maintenance process through replicated, controlled experiments." Journal Of Systems And Software **44**(3): 187-197.

von Mayrhauser, A. and A. M. Vans (1994). Comprehension processes during large scale maintenance. Proceedings of the 16th international conference on Software engineering. Sorrento, Italy, IEEE Computer Society Press.

von Mayrhauser, A. and A. M. Vans (1996). "Identification of Dynamic Comprehension Processes During Large Scale Maintenance." IEEE Trans. Softw. Eng. **22**(6): 424-437.

von Mayrhauser, A. and A. M. Vans (1997). Program understanding behavior during debugging of large scale software. Papers presented at the seventh workshop on Empirical studies of programmers. Alexandria, Virginia, United States, ACM Press.

Wiedenbeck, S., et al. (1999). "A comparison of the comprehension of object-oriented and procedural programs by novice programmers." Interacting With Computers **11**(3): 255-282.

Woodfield, S. N., et al. (1981). The effect of modularization and comments on program comprehension. Proceedings of the 5th international conference on Software engineering. San Diego, California, United States, IEEE Press.

Ye, N. and G. Salvendy (1996). "An objective approach to exploring skill differences in strategies of computer program comprehension." Behaviour & Information Technology **15**(3): 139-147.

Young, P. (1996). Program Comprehension, Visualisation Research Group, Centre for Software Maintenance, University of Durham.

Appendix A – Tables of Strategies and Developer Classification

Initial strategies identified by profiles

Subject_id	Strategy
1	Systematic
2	Systematic
3	As-needed
4	Systematic
5	Systematic
6	Systematic
7	Systematic
8	Systematic
9	As-needed
10	Systematic
11	Systematic
12	Systematic
13	Systematic
14	Systematic
15	Systematic
16	As-needed
17	Systematic
18	As-needed
19	Systematic
20	As-needed
21	As-needed
22	Systematic
23	Systematic
24	Systematic

Developer Classification

Subject_id	Expertise
1	Java Expert
2	Java Expert
3	Java Expert
4	Java Expert
5	Java Expert
6	Java Expert
7	Java Expert
8	Procedural Expert
9	Procedural Expert
10	Procedural Expert
11	Java Expert
12	Java Expert
13	Java Expert
14	Java Expert
15	Novice
16	Novice
17	Novice
18	Novice
19	Novice
20	Novice
21	Novice
22	Novice
23	Java Expert
24	Procedural Expert

Appendix B – Background Questionnaire

This background questionnaire is in Norwegian due to that the participants were all Norwegians. The questionnaire is not shown in the web-based form with text fields, radio buttons, etc., but in plain text:

Thinkaloud-replikeringseksperiment: Bakgrunnsskjema
Velkommen til Feedback Collection eksperimentet!

Dette eksperimentet er delt inn i to hoveddeler:

• Bakgrunnsinformasjonsskjema
• Feedback Collection eksperimentet

Formålet med bakgrunnsinformasjonsskjemaet er å hente inn informasjon som er relevant for selve analysen av eksperimentet.

Formålet med eksperimentet er å utforske forståelsen av objektorienterte konsepter.
Formålet med eksperimentet er <u>ikke</u> å evaluere hvor flink du er til å programmere.

I dette eksperimentet skal du løse endringsoppgaver på små applikasjoner i Java ved hjelp av JBuilder. For å forstå hvordan du løser oppgavene, må vi "titte" litt i tankene dine. Du vil bli bedt om å skrive hvordan du har tenkt i løpet av eksperimentet ved at du noterer ned tankene dine inn i et eget vindu, som vil dukke opp med jevne mellomrom. Dette kommer vi nærmere inn på senere.

Selve eksperimentet består av fem deler:
1. En øvelsesoppgave.
2. En oppgave som du skal gjennomføre på vanlig måte.
3. Oppvarmingsøvelser hvor du skal øve på å bruke Feedback-collection-vinduet sammen med en instruktør.
4. Tre oppgaver til. Du skal skrive kommentarene i Feedback-collection-vinduet mens du jobber.
5. Et spørsmålsskjema.

Før du starter på eksperimentet må du fylle ut informasjon om din bakgrunn.

Innledning bakgrunnsskjema

I dette spørreskjemaet skal du svare på spørsmål og påstander om din erfaring, utdanning og kompetanse.

Det er viktig at du svarer ut fra din egen mening og ikke ut fra hva andre måtte mene, eller det du tror er "riktig" svar. Det er ingen riktige eller gale svar. Det riktige svaret er det du selv mener passer best.

Ikke tenk for mye på hver påstand, men velg det som virker umiddelbart riktig. Det første innfallet er oftest det man egentlig mener.

Opplysningene vil utelukkende bli brukt som en del av eksperimentet, og vil bli behandlet konfidensielt.

Dette skjemaet må være ferdig utfylt før du kan begynne med selve eksperimentet.
Spørsmål merket med * må besvares!

Erfaringsskjema

3.1) **Personinformasjon**

3.1.1) *Fødselsår
Angi hvilket år du er født. (åååå)

3.1.2) *Kjønn
Mann Kvinne

3.2) **Arbeidserfaring**

3.2.1) *Hvor mange års arbeidserfaring med programmering/systemutvikling har du ?
Beskriv det gjerne utfyllende ved å ta med hvilket språk og når.

3.2.2) *Hvor mange års arbeidserfaring har du totalt?
Angi med ca. desimaltall.
Beskriv også gjerne hvilken stilling og årsperiode.

3.3) **Utdanning**

3.3.1) *Hvor mange vekttall har du totalt (20 vekttall = 1 år fulltids utdanning; 1 vekttall = 3 studiepoeng)?

3.3.2) *Hvor mange vekttall programmeringsrelatert informatikk har du?

3.4) **Kurs**

Kurs du har tatt privat eller interne/eksterne kurs gjennom jobben.
3.4.1) *Angi hvor mange datarelaterte kurs du har tatt (som ikke har gitt vekttall).
Husker du ikke eksakt så angi et cirka tall. Antallet trenger strengt tatt ikke å stemme med antall kurs du beskriver i neste spørsmål.

3.4.2) *Beskriv kurs du har tatt innenfor programmeringsrelatert informatikk og systemutvikling, og når (årstall).
Ikke ta med fag/kurs fra skoler som gir vekttall. Dette skal du beskrive senere.
Beskriv hva kurset gikk ut på, når og om det ble tatt internt/eksternt på jobben eller på privat basis.

Programmeringskompetanse

4.1) Generell programmeringskompetanse
4.1.1) *Hva er din vurdering av hvor dyktig du er som programmerer?

1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.2) Spesifikk programmeringskompetanse – Java

4.2.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.2.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.2.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.3) Spesifikk programmeringskompetanse - C++

4.3.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.3.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.3.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.4) Spesifikk programmeringskompetanse - C#

4.4.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.4.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.4.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.5) Spesifikk programmeringskompetanse - Simula

4.5.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.5.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.5.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.6) Spesifikk programmeringskompetanse - SmallTalk
49

4.6.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.6.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.6.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.7) Spesifikk programmeringskompetanse - Pascal

4.7.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.7.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.7.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.8) Spesifikk programmeringskompetanse - Python

4.8.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.8.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.8.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.9) Spesifikk programmeringskompetanse - C

4.9.1) *Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.9.2) *Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.9.3) *Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.10) Spesifikk programmeringskompetanse - Annet språk I

4.10.1) Angi navnet på språket

4.10.2) Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.10.3) Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.10.4) Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

4.11) Spesifikk programmeringskompetanse - Annet språk II

4.11.1) Angi navnet på språket

4.11.2) Hvor mange måneders arbeidserfaring har du totalt med dette programmeringsspråket?

4.11.3) Estimer omtrentlig hvor mange linjer kode du har programmert i dette språket:

4.11.4) Hva er din vurdering av hvor godt du kan dette programmeringsspråket?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

51

Verktøykjennskap

5.1) Erfaring med JBuilder

5.1.1) *Hvor mange års erfaring har du totalt med JBuilder?

5.1.2) *Hvor godt mener du at du kan JBuilder?
1 = Kan svært lite/ingenting, 5 = Ekspert
1 2 3 4 5

5.2) Erfaring med annet verktøy.

5.2.1) *Hvilke andre verktøy har du brukt og kjenner?
(Kryss av flere alternativer)
Eclipce
JBoss
Visual Cafe
Visual Age
Together
IBM Rational
JEdit
Visual J++
JCreator
SUN J2SDK
GNU Emacs
OptimalJ
Textpad
Annet, spesifiser:

5.2.2) *Skriv ned de tre verktøyene du har brukt mest og kjenner best, og hvor lang erfaringstid du har med hver enkel?
(Grader erfaringen din med verktøyet ved å angi et tall mellom 1-5 hvor 1 er lite og 5 er mye)

Designmetoder/ -notasjoner og pattern

6.1) Erfaringsskjema - Hvilke designmetoder/-notasjoner kjenner du til? (gradert fra 1-5)
(1 = Kan svært lite/ingenting, 5 = Ekspert)

6.1.1) *UML
1 2 3 4 5

6.1.2) *Data-driven design (relasjonsdatabaser eller lignende).
1 2 3 4 5

6.1.3) *OMT (Object modelling technique)
1 2 3 4 5

6.1.4) *Responsibility-driven design
1 2 3 4 5

6.1.5) *Strukturert analyse og/eller strukturert design.
1 2 3 4 5

6.1.6) *Rollemodellering
1 2 3 4 5

6.1.7) *Design patterns
1 2 3 4 5

6.1.8) *Arkitektur patterns
1 2 3 4 5

6.2) Andre designmetoder/-notasjoner du kjenner til II

6.2.1) Navn:

6.2.2) Kjennskapsgradering
(1 = Kan svært lite/ingenting, 5 = Ekspert)
1 2 3 4 5

Jobbfunksjon

7.1) *Hva er din primære jobbfunksjon i dag?
(Kryss av flere alternativer dersom du har mer enn én primærfunksjon)
Kravhåndtering / kravanalyse
Arkitektur og design
Koding / programmering
Testing

Kvalitet- og prosessutvikling
Vedlikehold
Prosjekt- /linjeledelse
Integrering (deployment)
Annet

7.2) Hvis du svarte "Annet på forrige spørsmål, spesifiser:

7.3) *Hva mener du er dine styrker innen systemutvikling?
(Kryss av for flere alternativer dersom dette er ønskelig)
Kravhåndtering / kravanalyse
Arkitektur og design
Koding / programmering
Testing
Kvalitet- og prosessutvikling
Vedlikehold
Prosjekt- /linjeledelse
Integrering (deployment)
Annet

7.4) Hvis du svarte "Annet på forrige spørsmål, spesifiser:

7.5) *Hvor mange OO-prosjekter har du deltatt i?

7.6) Hvis du har vært leder for OO prosjekter, angi hvor mange måneder du har vært dette totalt, og for hvert enkelt prosjekt.

7.7) Hvor mange ansatte hadde du under deg i hvert av prosjektene?
(List opp prosjektene og angi cirka antall ansatte i prosjektene du listet opp i forrige spørsmål).

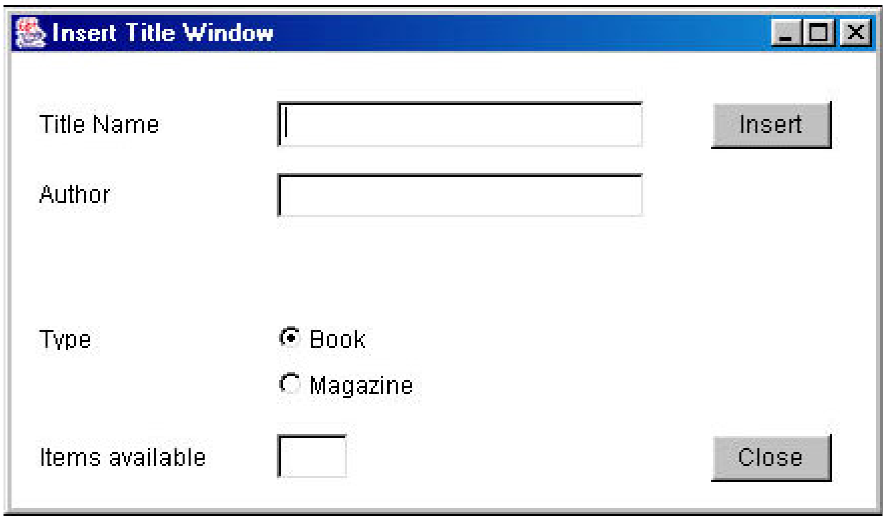
Avslutning
Takk!
Bakgrunnsskjemaet er nå ferdig utfylt.

Trykk på "Fullfør"-knappen for å avslutte denne sesjonen og vent på en bekreftelse. Når du har mottatt bekreftelsen trykker du på linken "Eksperiment" på menyen øverst (ved siden av "Logg av"-knappen) eller skriver adressen til SESE på nytt i nettleseren (www.sese.no) for å komme til neste del av eksperimentet.

Appendix C – Tasks and Detailed Task Description

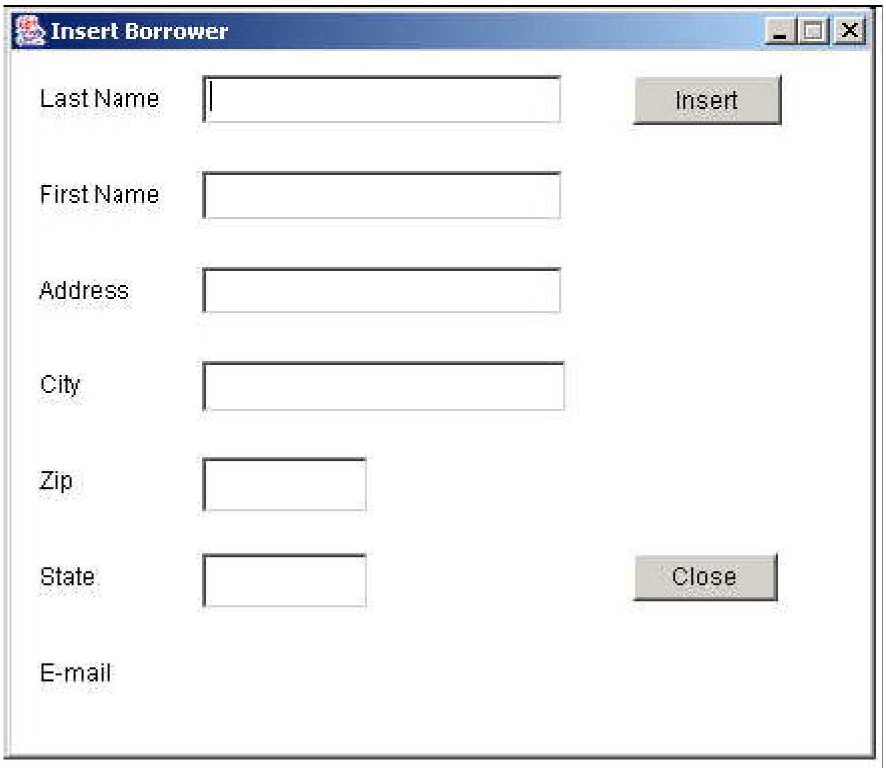
Task 2

Until now the ISBN-numbers have been stored in the system. A new international system for categorisation has been accepted, so this information is no longer needed. You must remove ISBN information from all the places where it has been used (you may comment it out). Also remember to remove all ISBN-text fields from the user interface. Here is an example on how one of the windows should look when you complete the task:



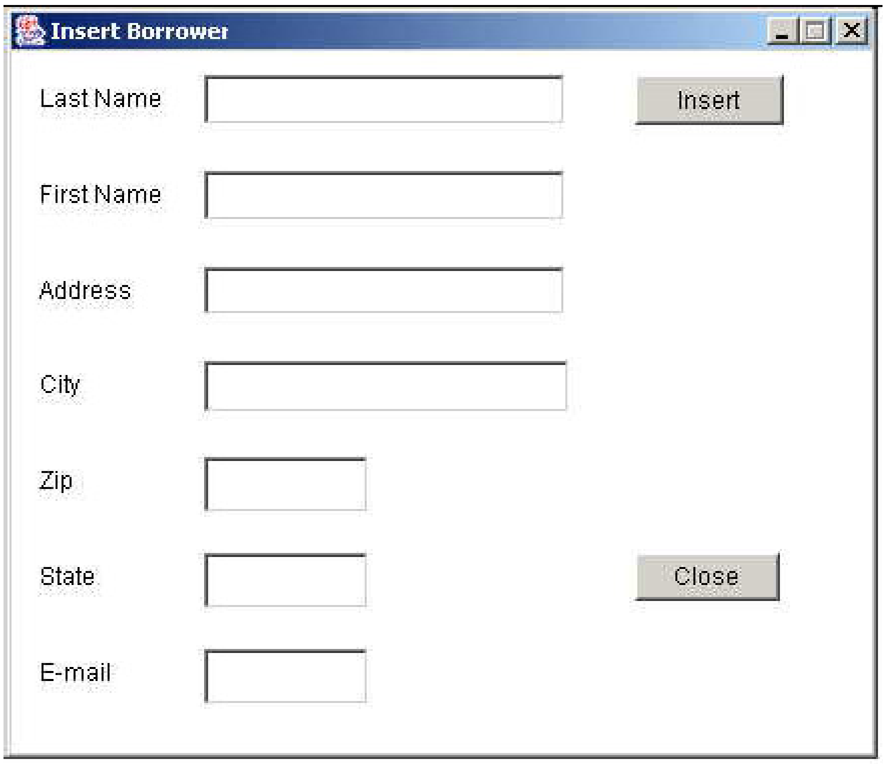
Task 3A

You are going to add the text “E-mail” to the window to insert new borrowers. You don’t have to create a new text field for reading the E-mail in this part of the task. The text “E-mail” should be placed under “State” as shown in the picture:



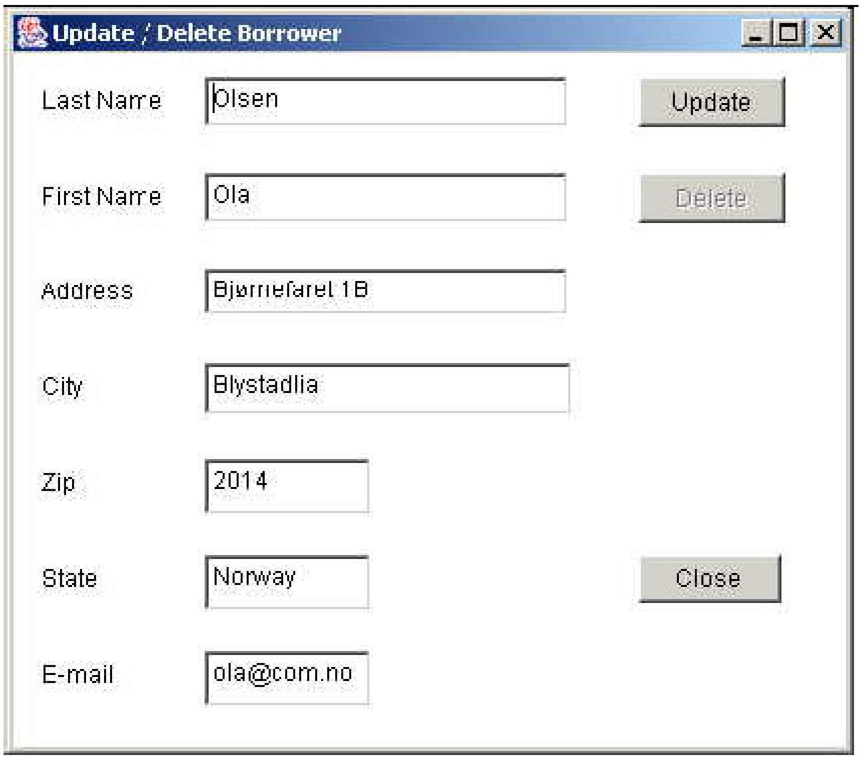
Task 3B

You should add a text field for E-mail to the class that contains data regarding borrowers. To write this field to the file and read from it you have to change the methods read() and write(). You must change the window for inserting borrowers so that E-mail can be entered. Remember to remove the *.dat-files before testing. The window should look like the screenshot below. You can test if you have written E-mail to the file by opening the .dat-file in Notepad.



Task 3C

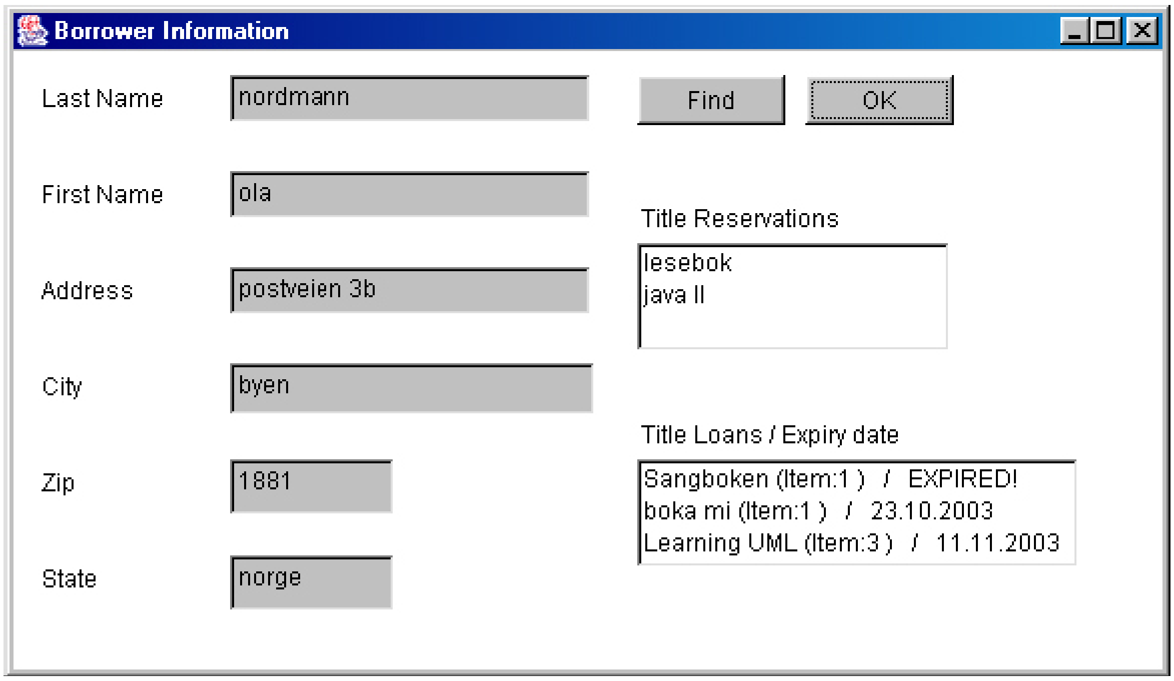
You should now change the window for updating borrower so that E-mail can be shown. The window should look like the screenshot below. Then you should change all other windows that show borrower's information in the same way.



Last Name	Olsen
First Name	Ola
Address	Bjørnefaret 1B
City	Blystadlia
Zip	2014
State	Norway
E-mail	ola@com.no

Task 4

You should add functionality for presenting the borrower loan due. The loan is due 4 weeks after the loan was made. You are going to show in the BorrowerInfoWindow-class that a borrower's loan has expired. An illustration of the window after new functionality has been added is shown below:


Borrower Information

Last Name	nordmann	
First Name	ola	
Address	postveien 3b	
City	byen	
Zip	1881	
State	norge	

Title Reservations

lesebok
java II

Title Loans / Expiry date

Sangboken (Item:1) / EXPIRED!
boka mi (Item:1) / 23.10.2003
Learning UML (Item:3) / 11.11.2003

You can use Java's Calendar-class http://java.sun.com/j2se/1.4.2/docs/api/java/util/Calendar.html) to handle dates (`java.util.Calendar`).

`Calendar.getInstance()` returns a GregorianCalendar-object where the time fields are initialized with date and time.

Some examples (you might need some methods in this task):

```
Calendar rightNow = Calendar.getInstance();
int year = rightNow.get(Calendar.YEAR);
int day = rightNow.get(Calendar.DATE);
int month = rightNow.get(Calendar.MONTH) + 1;

// Subtract 5 days from the date
rightNow.add(Calendar.DATE, -5);

// Returns true when rightNow is earlier than baseCal
boolean b = rigthNow.before(baseCal);

// Set the Calendar to a specific date (year, month, day,
hour, minute, second)
rightNow.set(2003, 7, 21, 10, 30, 30);
```

Detailed Task Description

Task 2

This was an alteration task which did not require the participant to make any addition of code, just remove some of the existing code. However, the application was rather large, and the task required the participants to make changes several places in the code. Although the task required changes in quite a few classes, it could be done simply by performing a search for ISBN through all the files (classes) and deleting the findings. From the total of 27 classes in the application, there were 5 classes containing ISBN.

Table 26 shows an overview of the classes that had to be altered in this task, with a description of what had to be done for each of the classes.

Table 26: Necessary alterations for Task 2

File name	Changes required
Title.java	This is the entity object in the BO-package where ISBN was stored. Changes needed on: - Constructor - Get/set methods - Read and write methods for persistency - A find method (if else)
FindTitleDialog.java	UI-package. Changes needed - Fields and labels with corresponding getText/setText methods - Declarations - FindButton_Clicked
TitleFrame.java	UI-package. Changes needed: - Fields and labels with corresponding getText/setText methods - Declarations - AddButton_Clicked
TitleInfoWindow.java	UI-package. Changes needed: - Fields and labels with corresponding setText method - Declarations
UpdateTitleFrame.java	UI-package. Changes needed : - Fields and labels with corresponding getText/setText methods - Declarations - UpdateButton_Clicked

Task 3
This task was divided into three parts to make it somewhat less complex for the participants:

The first part was merely to add a piece of text to the user interface. A screen shot of the window to be altered was given in the task text, so the participants could easily see in which class to do the editing.

The second part was a bit more complex. The participants should add an input field to the UI-class they had already altered in the former part. Also, they had to alter two methods in the entity class – read and write - to make sure that E-mail could be saved to file. They were given a hint to which class they had to alter ("the class containing information about a borrower"), and also which methods to alter ("read" and "write").

The third and last part was the most extensive of the three parts. The participants had to alter all the windows in the user interface which contained information about E-mail. However, the alteration task itself was similar to the one performed in the previous parts of this task. Table 27 shows an overview of the alterations that had to be made.

Table 27: Necessary alterations for Task 3

File name	Changes required
BorrowerInformation java	This is the entity object in the BO-package where E-mail should be stored. Changes needed on: • Constructor • Get/set methods • Read and write methods for persistency
BorrowerInfoWindow.java.	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations
BorrowerFrame.java	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations
FindBorrowerDialog.java	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations
UpdateBorrowerFrame.java	UI-package. Changes needed: • Fields and labels with corresponding getText/setText methods • Declarations

Task 4
This task was the most complex of the three. The participants had to introduce a new variable containing the date of a loan, and this variable had to be saved and loaded from file. The participants were given no hints as to where to place this variable (in which class). They also had to change the user interface, calculate an expiry date (4 weeks after the loan was made)

and include logic to check the expiry date against today's date. They were given the name of the class in which to do this. What also complicated this task was that they had to use a class – Calendar – which was previously unknown for most of the participants. Some information on the usage was given in the task text, but not all the methods required solving the task.

Table 28 shows an overview of the alterations that had to be made during Task 4.

Table 28: Necessary alterations for Task 4

File name	Changes required
Loan.java	This is the entity object in the BO-package where the loan date should be stored. Changes needed on: • Constructor • Get/set methods • Read and write methods for persistency • Logic for calculating expiry date
BorrowerInfoWindow.java	UI-package. Changes needed: • Method for displaying the expiry date in the window

Appendix D – The Application

The sections below contain an overview of the classes in each package.

User Interface Overview
The User Interface package is on top of the other packages. It presents the services and information in the system to the user. The package cooperates with the business objects package. The User Interface package calls operations on the business objects to retrieve and insert data into them. See Table 29 for an overview of the classes in this package.

Table 29: Classes in the User Interface package

Class name
AboutDialog.java
BorrowerFrame.java
BorroweInfoWindow.java
BrowseWindow.java
CancelReservationFrame.java
FindBorrowerDialog.java
FindTitleDialog.java
LendItemFrame.java
MainWindow.java
MessageBox.java
QuitDialog.java
ReservationFrame.java
ResultOfFindBorrower.java
ResultOfFindTitle.java
ReturnItemFrame.java
TitleFrame.java
TitleInfoWindow.java
UpdateBorrowerFrame.java
UpdateTitleFrame.java

Business Objects Overview
This is the business objects package in the design. All business object classes inherit the Persistent class in the Database package - which is described in the next section. Each class has its own read and write method, but it is the services from the Database package that allows the classes to be stored persistently. See Table 30 for an overview of the classes in this package.

Table 30: Classes in the Business Object package

Class name	Description
BorrowerInformation.java	Information about a borrower. A borrower can be a person or another library. A borrower can have loans and reservations.
Item.java	Represents an item, a physical instance of a Title. The library can have several items of one title.

Loan.java	Represents a loan. The loan refers to one title and one borrower.
Reservation.java	A reservation reserves a title for a specific borrower, meaning the borrower should be first in line when any item of the title is returned.
Title.java	Represents a book or magazine title. A title can exist in many physical items and can have reservations connected to it.

Database Overview
The Database package is necessary to provide persistent storing of the objects in the Business Object package. All the implementation of persistent storage handling is done in a class called Persistent, which classes that needs persistent objects must inherit. See Table 31 for an overview of the classes in this package.

Table 31: Classes in the Database package

Class name	Description
Persistent.java	Super class for classes that wants to be persistent. Subclass must inherit read and write operations that serializes the object to disk. This class reads files sequentially.

Utility Overview
The Utility package contains only one class – ObjId, which is used to refer to persistent objects throughout the system. The class is used both in the user-interface, business object, and database package. See Table 32 for an overview of the classes in this package.

Table 32: Classes in the Utility package

Class name	Description
ObjId.java	Represents an object id, a class that works as “pointer” to any persistent object in the system.