Comprehension strategies and difficulties in maintaining object-oriented systems: an explorative study

Amela Karahasanović [1], Annette Kristin Levine [1], Richard Thomas [2]

amela@simula.no, annettel@simula.no
richard@csse.uwa.edu.au

[1] Simula Research Laboratory
P.O. Box 134,
NO-1325 Lysaker, Norway

[2] School of Computer Science & Software Engineering,
M002, The University of Western Australia,
Crawley, Western Australia 6009, Australia

Abstract. Program comprehension is a major time-consuming activity in software maintenance. Understanding the underlying mechanisms of program comprehension is therefore necessary for improving software maintenance. It has been argued that acquiring knowledge of how a program works before modifying it (the systematic strategy) is unrealistic in larger programs. The goal of the experiment presented in this paper is to explore this claim. The experiment examines strategies for program comprehension and cognitive difficulties of developers who maintain an unfamiliar object-oriented system. The subjects were 38 students in their third or fourth year of study in computer science. They used a professional Java tool to perform 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 were more likely to produce correct solutions. Two major groups of difficulties were related to the comprehension of the application structure, namely to the understanding of GUI implementation and OO comprehension and programming. Acquisition of strategic knowledge might improve program comprehension in software maintenance.

Keywords: Maintenance; Program comprehension; Experiment; Object-oriented

1. Introduction

It is widely accepted that software maintenance absorbs a significant amount of the effort expended in software development. Although the reported figures differ, most researchers on software maintenance agree that more than 50% of programming effort is constituted by changes made to the system after the implementation (Coleman *et al.*, 1994; Holgeid *et al.*, 2000; Lehman and Belady, 1985; Lientz, 1983; Nosek and Prashant, 1990; Pfleeger, 1987; Zelkowitz, 1978). Program comprehension has been

recognised as a major time-consuming process in software maintenance (Storey *et al.*, 1999) taking up to 60% of the total time devoted to maintenance (De Lucia *et al.*, 1996; Dunsmore *et al.*, 2000).

Object-oriented (OO) programming has become a *de facto* standard and therefore we need to understand the problems of maintaining such systems, including comprehension strategies. Object-oriented programming is increasingly being taught in computer science courses. A survey conducted by Dale (2005a; 2005b) shows that 65% of the participating educational institutions teach object-oriented programming as a part of introductory courses in computer science education. Some of these students are going to have to maintain object-oriented systems when they start work. It is therefore important to understand the comprehension strategies they use while maintaining such systems. It is a prerequisite for the development of tools, documentation and maintenance guidelines that support their cognitive processes in an appropriate manner and thus improve maintenance. Furthermore, understanding the cognitive difficulties of students can enable educators to improve their teaching.

Numerous studies have been performed in the area of program comprehension over the last few decades (Détienne, 1997; Détienne, 2002; Storey, 2005; Upchurch, 2002). Research has been conducted in the contexts of general strategies (Burkhardt *et al.*, 1998; Mosemann and Wiedenbeck, 2001; O'Brien and Buckley, 2001; Von Mayrhauser and Vans, 1996; Wiedenbeck and Ramalingam, 1999; Wiedenbeck *et al.*, 1999), software maintenance and enhancement (Corritore and Wiedenbeck, 2001; Parkin, 2004; Pennington, 1987a; Von Mayrhauser and Vans, 1997), reuse and documentation (Burkhardt *et al.*, 2002), and debugging (Ko and Uttl, 2003).

Although much research has been done, Corritore and Wiedenbeck (2001) point out that our understanding of program comprehension remains incomplete and that there is a need for deeper knowledge of the comprehension strategies employed during software maintenance tasks. Research has been done on characterising the strategies employed by different kinds of programmers, but little is known about *why* these particular strategies emerge (Davies, 1993). Furthermore, experiments on program comprehension have been criticised for their lack of realism. Programs used in these experiments are typically small; the largest reported program is about 822 lines of code (LOC) (Corritore and Wiedenbeck, 1999; Corritore and Wiedenbeck, 2001). 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). Furthermore, the participants in experiments have been given the program to study for a period of time, followed by maintenance tasks or questionnaires (Pennington, 1987a; Corritore and Wiedenbeck, 2001). However, in practice, the required maintenance tasks are usually known before starting comprehension, and so comprehension is driven by the maintenance tasks. Most experimental designs do not reflect this. Furthermore, rather less attention has been paid to interactions between the strategies that programmers use and the difficulties they have while comprehending object-oriented programs.

The study reported in this paper aims to add to the body of knowledge on program comprehension by exploring strategies and difficulties of programmers when conducting maintenance tasks in a more realistic context. The program used in this study was a 3600 LOC large library application system written in Java. The participants were 38 students in their third or fourth year of study in computer science

at either the University of Oslo or Oslo University College, Norway. The experiment lasted six hours and the participants conducted two maintenance tasks on the given application system, using JBuilder. The participants were provided with documentation that describes the application system and the JBuilder documentation. They had access to the Java online documentation and could use their Java books. In addition, both the program and the tasks were presented at the same time, leaving it up to the participants whether or not they wanted to familiarise themselves with the program before proceeding with performing the tasks.

The remainder of this paper is organised as follows. Section 2 describes related work. Section 3 describes the methodology. Section 4 presents the results. Section 5 discusses threats to validity. Section 6 concludes and suggests avenues for further work.

2. Strategies for comprehension of object-oriented systems

Much research has been conducted in the area of program comprehension over the last few decades. Détienne reviews cognitive models and experiments conducted in this area (Détienne, 2002) and provides a survey of empirical studies on object-oriented design (Détienne, 1997). Storey (2005) reviews some of the key theories of program comprehension and discusses how these theories are related to tools that support it. Upchurch (2002) gives an extensive bibliography of the field.

Section 2.1 describes the concepts underlying our research. Section 2.2 surveys related empirical studies. Section 2.3 states our research questions.

2.1. Comprehension strategies

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" (Corritore and Wiedenbeck, 2001).

During the program comprehension, developers build their own mental representation of the program to be understood; a mental model. The cognitive processes of developers and temporary information structures they use to develop their mental model are described by a cognitive model of program comprehension.

Existing cognitive models can be classified as top-down, bottom-up or a combination of these two. According to the top-down (domain) model of program comprehension (Brooks, 1983; Soloway *et al.*, 1988) programmers start with a general hypothesis about the overall purpose of the program. Each component is then viewed in the light of this hypothesis. In the bottom-up model (Pennington, 1987a; Pennington, 1987b; Shneiderman and Mayer, 1979) programmers start by reading code statements and group these statements until a high-level mental representation of the program is constructed. Pennington (1987a; 1987b) describes two program abstractions that are formed by the programmer during comprehension; the *program model*, which is a low-level abstraction, and the *domain model,* which is a higher level of abstraction. She also described four basic categories of program information

making up the programmer's mental representation: elementary operations in the code, control flow, data flow and program goals. Von Mayrhauser and Vans have proposed the Integrated Metamodel of program comprehension (Von Mayerhauser and Vans, 1993; Von Mayerhauser and Vans, 1995) based on the above-described models. It consists of four components: the top-down, situation and program models, and the knowledge base. The program, situation and domain models describe comprehension processes used to create mental representations of the program. The knowledge base describes the knowledge needed to perform these processes.

A comprehension strategy is a high-level approach to comprehension (Von Mayrhauser and Vans, 1996). It is a method applied by developers to understand a given program. The direction of the comprehension strategy concerns the programmer's approach to program comprehension regarding abstraction levels (top-down, bottom-up and combined). The breadth of the comprehension strategy concerns the programmer's approach regarding the scope of comprehension. Littman and his colleagues (1986a) observed two strategies regarding breadth of comprehension: systematic and as-needed. The systematic strategy involves an attempt to gain knowledge and understanding about a program before carrying out modifications, whereas the as-needed strategy is used by programmers who minimize the amount of code to be understood before modifying the program. They observed that programmers who used the systematic approach carried out modifications more successfully than those with an as-needed approach. The reason, they argue, is that systematic study of the program increases the ability to detect interactions among its components. Koenemann and Robertson (1991) argue that the systematic approach is unrealistic in larger programs. The program used in the Littman experiment was only 200 LOC. Koenemann and Robertson conducted an experiment in which a group of expert programmers carried out modifications on a 600 LOC program. None of the programmers attempted to use a truly systematic strategy. The scope was directed by the modification task they performed, and they did not try to comprehend parts that did not have a direct bearing on this task. One instance of the systematic strategy during the comprehension of a large-scale program has been reported by Von Mayrhauser and Vans (1996). All these studies were conducted in the procedural paradigm.

2.2. Empirical studies of comprehension in the object-oriented paradigm

Studies on object-oriented comprehension have explored different aspects of cognitive models. One line of research focuses on the effects on information structures that the developers use in understanding object-oriented programs of the following factors: paradigm (Corritore and Wiedenbeck, 1999; Corritore and Wiedenbeck, 2001; Wiedenbeck and Ramalingam, 1999; Wiedenbeck *et al.*, 1999; Khazaei and Jackson, 2002), expertise (Burkhardt *et al.*, 2002; Corritore and Wiedenbeck, 1999; Corritore and Wiedenbeck, 2001), view, such as, data flow, sequential, control (Mosemann and Wiedenbeck, 2001), tasks (Burkhardt *et al.*, 2002) and phase (Burkhardt *et al.*, 2002; Corritore and Wiedenbeck, 1999; Corritore and Wiedenbeck, 2001).

The second line of research focuses on the comprehension strategies of developers. Table 1 summarises studies of comprehension strategies in the object-oriented paradigm regarding participants, applications (language and size), tasks (type of task and duration), environments (e.g. pen and paper or computer) and method of data collection.

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

Study	Participants	Application Tasks Environment	Data collection method
(Burkhardt *et al.*, 1998)	49 professionals (28 OO experts, 21 procedural experts)	C++ 550 LOC Comprehension for later documentation or reuse 35 min. program study	Verbal protocols
(Corritore and Wiedenbeck, 1999; Corritore and Wiedenbeck, 2001)	30 professionals (15 OO experts, 15 procedural experts)	C and C++ 783 and 822 LOC Maintenance tasks Two 2-hour sessions (a week between them) Code and documentation available on-line; only one document visible at a time	Screen capture software, Comprehension questionnaire (used for research on information strategies)
(Torchiano, 2004)	Explorative study 28 4th year students	Java 628 LOC Duration not reported Maintenance tasks Code and documentation available on-line	User action capture software

Numbers of participants in these studies ranged from 28 to 49. The participants were both students and professional developers. The size of applications (programs the participants needed to comprehend) used in these studies ranged from 550 to 822 LOC and the applications were written in C, C++ or Java. The participants were asked to study the given applications (general comprehension) or to perform some maintenance, documentation or reuse tasks. The time the participants were given to perform given tasks ranged from about 35 minutes to two two-hour sessions with a week between them. The participants were either given a hard copy of the program to study (pen and paper exercises) or had the code and documentation available on-line on their computers. The data were collected by means of verbal protocols, screen capture software, comprehension questionnaires, and user action capture programs. Their comprehension strategies were identified by the following measures:

- the number of different files they visited (obtained from the verbal protocols or screen capture software)

- the number of documentation pages they visited and the time they spent on those pages (obtained from the user actions capture software)

Burkhardt et al. (1998) studied the effects of experts' comprehension strategies in the object-oriented paradigm. They found evidence of top-down behaviour in expert comprehension. By contrast, novices used a more bottom-up strategy and their comprehension was execution-based. They also found that the experts tended to consult more files.

A study conducted by Corritore and Wiedenbeck (2000; 2001) examined comprehension strategies of experts in two paradigms while conducting maintenance tasks. Their results suggest that during the early phase of program comprehension the object-oriented programmers used a strong top-down approach, but that during the maintenance tasks they used a bottom-up approach. The procedural programmers started with a bottom-up approach and used it even more during the maintenance tasks. The breadth of comprehension, i.e. the extent to which the programmer becomes familiar with all or most parts of program during comprehension, was found to be greater for the procedural than for the object-oriented programmers. However, results suggest that expert programmers build a broad and systematic, rather than a localized, view of a program after conducting several maintenance tasks. The authors argue that object-oriented programmers focus on documentation early in comprehension, as the most pertinent information sought, namely domain objects and their relationships, is clearly presented in object-oriented documentation. The results of Torchiano (2004) suggest that pattern-specific documentation supports both top-down and bottom-up comprehension.

The applications used in these studies are small by industry standards and it is possible that the results are not applicable to situations in which the developers comprehend and maintain larger applications. It is also possible that the results are not applicable to situations in which the developers work with professional development tools.

The third line of the research focuses on cognitive consequences of the object-oriented approach. Object-oriented systems are claimed to have several advantages: naturalness (Meyer, 1988), ease of use (Rosson and Alpert, 1990), reusability (Johnson and Foote, 1988) and maintainability (Henry and Humphrey, 1993; Daly et al., 1996). It has been argued that the world can be naturally structured in terms of objects, and that mapping between the problem domain and the programming domain is thus more straightforward in object-oriented paradigm (Meyer, 1988). Object-oriented systems might be particularly valuable in new domains or for experienced designers (Rosson and Alpert, 1990). Furthermore, reuse of software is easy in object-oriented systems as class hierarchies are well suited for reuse (Johnson and Foote, 1988).

Détienne (1997) surveyed empirical studies validating these claims in the context of object-oriented software design. The literature on experts supports the claims about naturalness and ease of OOD for experts. The literature on novice OO designers shows that they had difficulties in process of class creation and with declarative and procedural aspects of their solutions. Studies in this survey support the claim that OO paradigm promotes reuse by inheritance, but also identify problems the novice designers have with reuse.

Henry and Humphrey (1993) found that the modifications to object-oriented systems were more local, i.e., involved editing of fewer modules. Brian et al. (1997) found that adhering to good object-oriented design practices provides ease of understanding and modification.

Studies have also been conducted to identify what the programmers find difficult in object-oriented development. A study conducted with graduate students (Tegarden and Sheetz, 2001) revealed that decomposition was the most important contributor to the complexity of object oriented development. This includes class design (encapsulation, intra- and inter-class complexity, proper placement of methods and attributes) and structure (relationship between classes, objects and instances). A study conducted with professional developers (Sheetz, 2002) revealed that novices were overwhelmed with implementation issues, whereas experts were more focused on project management issues.

The above described studies validated claims on advantages of object-oriented paradigm and identify some of its difficulties. However, little has been known about difficulties during the maintenance. Furthermore, there are no studies that explore interactions between the strategies for comprehension that programmers use and the difficulties they have while attempting to comprehend object-oriented programs. In order to improve program maintenance practice, a deeper understanding of these interactions is needed.

2.3. Research questions

The goal of the research reported here was to add to the existing body of knowledge regarding object-oriented program comprehension by exploring strategies used by advanced beginners when conducting maintenance tasks on medium-sized applications. We had the following research questions:

RQ1 Do advanced beginners perform better if they use the as-needed strategy, rather than the systematic strategy? It is expected that the systematic strategy scales badly and is therefore not appropriate for maintaining medium-size object-oriented programs. If so, we should expect the participants using this strategy to perform worse in terms of time and correctness.

RQ2 Is there any relationship between strategies the advanced beginners use and the difficulties they have while maintaining medium-size object-oriented programs? As the systematic strategy provides an overview of the application, we would expect the programmers using this strategy to have fewer problems in understanding the structure of the application.

RQ3 Why do advanced beginners choose the particular strategy that they use? As they have some programming experience, we would expect them to choose a strategy that is appropriate for the given tasks, application and environment.

3. Design of the experiment

We conducted a controlled experiment with two goals:

- evaluating an experience sampling method, called feedback collection method, regarding its usefulness in comprehension studies
- exploring the participant's strategies and problems while they were conducting maintenance tasks on an object-oriented application.

The results regarding the first goal are reported in Karahasanovic *et al.* (2004). The feedback-collection method (FCM) was compared with concurrent think-aloud (CTA) and retrospective think-aloud (RTA) regarding type and usefulness of the collected information, costs related to analysis of the collected information, and effects of the data collection methods on the subjects' performance. The results showed that FCM allowed us to identify a greater number of comprehension problems that prevented progress or caused significant delay. It was less precise in identifying strategies for comprehension than CTA. FCM was less expensive in analysis (transcription and coding) than the other two methods. The results indicate that all three methods of data collection were intrusive and affected the performance of the subjects with respect to time and correctness (small to medium effect size).

This paper reports the results regarding the second goal.

3.1. Participants and setting

The participants were 38 students in their third or fourth year of study in computer science at either the University of Oslo or Oslo University College, Norway. They were asked to volunteer for a day during a vacation and were paid. The mean age was 24, range 20-38, with one female. All of the participants had been taught object-oriented programming and the Java programming language throughout their university courses. They had taken between four to 30 credits of such courses, the median being 10 (one course corresponds to three or four credits; a full school year is 20 credits). The participants reported they that had produced between 1000 and 50000 (median 7000) lines of Java code throughout their university courses or while working in industry. The participants' self-reported knowledge of JBuilder was medium (median 3, on a five-point scale: 1 means no experience with JBuilder, 5 means expert). Nineteen participants had industrial experience with Java programming, ranging from 2.5 months to 3.5 years, the median being one year. Thirty-nine participants started with experiment. However, one person dropped out towards the end of testing due to major technical problems, which left 38 participants for the main experiment. His background was average.

These participants attended an experimental session at Simula Research Laboratory, outside their educational institutions. They started in either the morning or afternoon and stayed for about six hours. The experiment was conducted on six separate days, due to restrictions pertaining to the use of required resources (individual rooms and observers). There were from four to 10 participants per day.

3.2. Treatments

The experiment was conducted in terms of four approaches: concurrent think-aloud (CTA), immediate retrospective think-aloud (RTA), feedback-collection method (FCM) and a control silent group (CS).

- The participants in the concurrent think-aloud group were instructed to verbalise their thoughts while performing the tasks. When a participant fell silent for about 30 seconds the observer used a reminder "keep talking". The observer was allowed to answer practical questions or questions related to a technical problem (not directly contributing to the task-solving process). Observers were instructed to have no other interaction with participants.
- The participants in the immediate retrospective think-aloud group were asked to provide an immediate retrospective report after each of the main tasks. The observers were instructed to ask open-ended questions with minimum prompting.
- For the feedback-collection group, a screen appeared every 15 minutes with the text "*What are you thinking now?*" The participants were instructed to describe what they were thinking just before the screen appeared. The time available for writing feedback was limited to two minutes. For the feedback-collection and control silent groups, an observer was present to help with technical problems. He was instructed to have no other interaction with the participants.

Each subject using think-aloud method (CTA or RTA groups) was located in an individual room and accompanied by an observer. Subjects working silently and using the feedback-collection method were accommodated in a laboratory of up to eight people with one observer. The participants were not allowed to talk with each other.

3.3. Data collection and supporting tools

Each participant used a PC running Windows. A remote connection was established to a Windows Server using Terminal Services Client. Thus, each participant saw a normal desktop, and could access Borland JBuilder for Java development.

A Web-based tool, the Simula Experiment Support Environment (SESE) (Arisholm *et al.*, 2002) was used for logistical support. The participants used this tool to answer the background questionnaire, to download the documents and code, to upload their solutions, and to provide feedback (FCM group only). The tool recorded the start-time and end-time for each task. Keystrokes, mouse-clicks and window focus events were logged with timestamps in milliseconds by the GRUMPS-Lite software (Thomas *et al.*, 2003). Each think-aloud session (concurrent and retrospective) was audio-recorded. Observers also made notes during the experiment.

3.4. Experimental procedure

Every participant first completed a training task. The next stage was to undertake a pretest task for up to an hour. A training session followed, in which participants in the feedback-collection method treatment practised it. After this initial training, the participants were asked to conduct three change tasks on a medium-size Java application, each intended to be more complicated and complex than its predecessor. The main session lasted from three to four hours. Each participant uploaded amended source code after each task, once he was satisfied with their performance. The participants were accommodated in a laboratory of up to eight people plus an observer.

Two weeks after the experiment, open-ended group interviews were conducted for about one hour with the participants, to provide additional information on their perception of the experiment. The participants were divided into eight groups. One interviewer and one observer conducted the interviews.

After several months, a presentation of the results was given to the teachers and students at the Oslo University College. They, in turn, presented their views on processes and problems regarding the comprehension of object-oriented methods.

3.5. Tasks

The participants were asked to complete a small training task and a pretest task. The purpose of the training task was to render subjects familiar with SESE and the experimental situation. The participants were to download the task, develop a Java program that writes a string in the reverse order, and upload their solutions. The pretask was to extend the functionality of a bank teller machine program. This application was a small Java program consisting of seven Java classes and about 400 lines of code (LOC). It was to be extended to provide a printout of all successfully performed transactions (deposits and withdrawals) for a given bank account. The purpose of the pretest task was to provide a basis for comparing the programming skill of the four treatment groups. These tasks were taken from (Arisholm *et al.*, 2001).

The tasks of the experiment pertained to the modification of 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 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. They can reserve a title if it is not available. The application consists of four packages with a total of 3600 LOC in 26 Java classes. This application system was used because we assumed that the application domain would be very familiar to the participants. The application system (UML diagrams and Java code) has been used as a case study in a UML textbook and can be considered as a good example of object-oriented practice. The participants were asked to conduct the following changes on the library application system:

Task1 *Delete functionality related to ISBN number.*
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. ISBN information should be removed from all the places where it has been used including the user interface.

Task2 *Extend the system to handle borrowers' e-mail addresses.*
The system should be extended to handle (read, store, change, display) borrowers' e-mail addresses. The window for inserting borrowers should be changed so that E-mail can be entered. The window for updating borrower should be also changed so that E-mail can be shown and updated. All other windows that show borrower's information should also be changed.

The tasks were ordered by complexity. The participants were provided with documentation describing the functionality and structure of the library application system and the JBuilder documentation. They also had access to the Java online documentation. Those subjects who managed to complete both change tasks on the library application within the allocated time were given an extra change task. This was to ensure that none of the subjects finished before the end of the allocated time for the experiment. We did not want any subject to disturb the other subjects by leaving early. This task was not included in the analysis. The participants were told clearly that they are expected to implement all specified functionality. The documentation given to the participants included test examples and figures (screen dumps) of the correct solutions.

3.6. Analysis model

To answer the above-stated research questions, we analysed quantitative data on performance and qualitative data from verbal protocols and interviews.

3.6.1 Breadth of comprehension

For the purpose of this analysis, we make the following definitions:

- A *systematic strategy* is one in which a participant tries to get an overview of the program by reading the documentation, reading the source code or running the application before he starts to make the required changes.
- An *as-needed strategy* is one in which a participant starts to perform the maintenance tasks without first trying to understand the system.

To identify comprehension strategies, two authors analysed the data collected by the user actions capture program (GRUMPS). The third author analysed verbal protocols and identified strategies from them. We then compared the findings. In 75% of cases the strategies identified from these two data sources were the same. The rest of the cases were resolved through more detailed analysis of the collected data and discussion between the authors.

Identification of strategies from user action logs

A class profile of a participant consists of an overview of classes visited by a participant, and the time spent in each class, ordered chronologically. This

information was extracted from the GRUMPS database. GRUMPS gave us the list of all the windows opened, identified the one that had the current focus and gave us the list of the actions performed by the participants in each window. If the participants performed some editing, their typing was recorded and could be traced back to the window.

The number of different files visited by the participants was used in the previous work (Koenemann and Robertson, 1991; Corritore and Wiedenbeck, 2001) to identify comprehension strategies. However, if a participant spent a very short period of time (only a few seconds) in a class, it is not possible to say that he was studying it. The class profiles showed two trends: either (i) the participants spent a longer time on the majority of classes they visited in the beginning, or (ii) they opened many classes quickly and then spent a longer time working with a class that needed to be edited. We assumed that the first group applied the systematic strategy, whereas the second applied the as-needed strategy. We consider that for the given application, 60 seconds is the minimum time that a participant would need to comprehend a class. Therefore we categorised the class profiles as follows: if a participant spent more than 60 seconds per visit for the majority of the classes visited during the first third of the time spent on the Task 1 they applied the systematic strategy, otherwise they applied the as-needed strategy. This categorisation was justified later on by the results from the verbal protocols.

Identification of strategies form verbal protocols

The verbal protocol data (concurrent think-aloud, retrospective think-aloud, and feedback-collection) was analysed to explore processes of program comprehension.

Information provided in the collected protocols was categorised according to a coding schema that we used in our previous research (Appendix A; Karahasanovic *et al.*, 2005). To identify strategies, we used information about the task-performing actions that the participants conducted (category 3.1 in the coding schema) or planned (category 3.3). We observed two different behaviours amongst the participants. The first group started by reading the documentation, reading the code or running the application before they conducted any changes. The other group started to change the code immediately. Based on this, the s*ystematic* strategy was identified by statements that indicated that a participant started by reading the documentation, reading the code or running the library applications. The *as-needed* strategy was identified by statements that indicated that a participant did not read the documentation, or that the participant quickly opened many classes and used the search function of JBuilder to find the places he should change. None of the participants explicitly reported that he opened many classes and started to edit without using the search function of JBuilder.

Data preparation and analysis were as follows. First, the audio-recorded data was transcribed verbatim. Following that, the collected data was analysed by one researcher to determine the general type of problem-solving strategy the participants had applied (explorative or systematic). Following the procedure given by (Ericsson and Simon, 1993), two transcripts for each treatment/strategy combination (12 in total) were chosen randomly to test a coding schema. The transcripts and collected feedback were coded by two researchers independently. We used semantic segmentation as recommended by Chi (1997). The protocols were divided in smaller units based on semantic features, such as ideas and topics, and then encoded. We

made no common segmentation of the protocols before coding. Encoded files were inspected for differences and questionable statements. Any problems identified were then resolved through discussion and analysis of their context. On the basis of an overall agreement of 91,4% between coders, the coding schema was judged to be reliable (Hughes and Parkes, 2003; Van Someren *et al.*, 1994) and applied to all transcripts and collected feedback (28 files in total [1]). The transcription and coding process took about 258 hours (about 168 hours for transcription of 28 hours of audio-recorded material; about 90 hours for coding and testing of the coding schema). In the examples presented in the following subsections, […] indicates an explanatory comment added by the authors. The examples were originally written in Norwegian and have been translated by the authors for this paper.

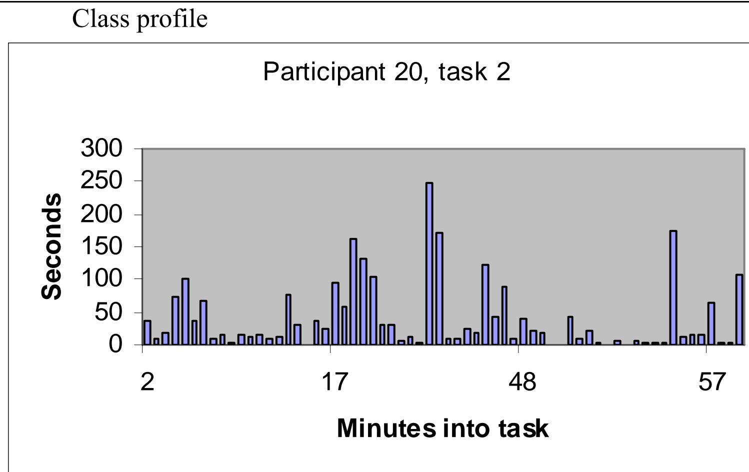
Examples of comprehension strategies

Tables 2 and 3 give examples of class profiles and the corresponding verbal protocols. The first class profile (Table 2) is representative of those participants who applied the systematic strategy. Such participants visited a relatively small number of classes and spent a relatively long time working on them early in the task-solving process. The verbal report confirmed that the participant tried to understand the application before he began to make the required changes. He ran the application to understand its functionality. He read the documentation, and tried to understand the structure. After having done all this, he started to modify the program.

[1] Three concurrent think-aloud protocols and one retrospective think-aloud protocol were not transcribed because the participants spoke very quietly and indistinctly.

Table 2
Example of the systematic strategy (Participant 20)

Class profile

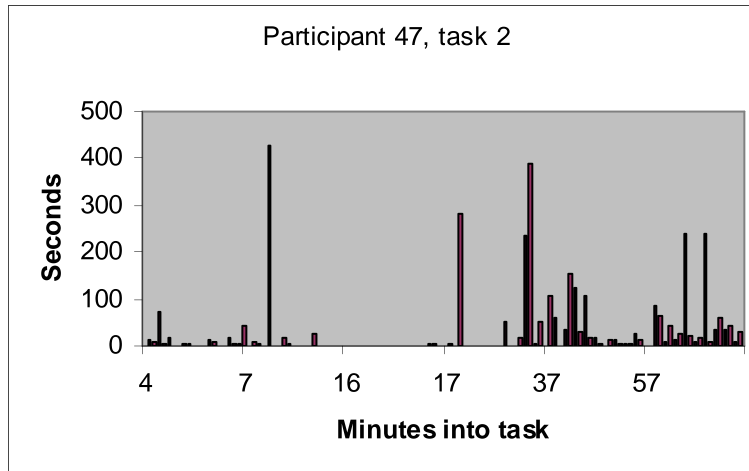


Excerpt from the verbal protocol (retrospective think-aloud)
So when I got the task, I thought – okay, this I have to try to understand, so I started to read the documentation, to the part... I read which classes were included. Then I thought, okay, this is a library system, a standard set up. Okay, I'll try. I tried to initiate a borrower and a book. I tried to lend the book. Okay, it worked well. Then I looked through the class diagrams, okay. … Then I understood better how it worked. But, okay, after that I looked at the task. And I saw what I was supposed to do. Just make changes in... remove the ISBN numbers, and then I thought that I should begin with the data part, on the data application layer.

The second class profile (Table 3) is representative of the as-needed strategy. This participant quickly opened many classes and then spent a longer time working on the classes that needed to be edited. The verbal report confirmed that the participant did not attempt to understand the application before he began to make the changes. Instead, he used JBuilder's functionality to perform the task.

Table 3
Example of the as-needed strategy (Participant 47)

Class profile



Excerpt from the verbal protocol (retrospective think-aloud)
My first thought, because I'm working on a tool like JBuilder which checks the dependencies in the code, I thought that it would be intelligent to find where these ISBN-fields were defined and then find the references from there. I went into the application to get an overview to where I might find these, and found out that it was the object Title which had the ISBN-field defined. And I deleted the field that defined the ISBN-field there, and got some error messages, and got a few references through that. And by doing this I managed to delete most of them. And then I went through the application and went through all the dialogues and found the last ones. That was the way I solved the task. I used the functionality of the program. I did not use the documentation.

3.6.2. Time

This is the time taken, 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/her solution, as recorded by the SESE tool.

These times were reconciled by the first and the last author against user action logs produced by GRUMPS. Lunch breaks or other breaks longer than 10 minutes were subtracted from task times. Thus, the variable *time* is the time that participants spent on the change tasks, excluding major nonproductive breaks.

3.6.3. Correctness

This is the assessment of the quality of the solution, which was marked on the basis of completeness. A solution was marked 'correct' if it worked and delivered all

functionality described in the given specification. Otherwise a solution was marked 'incorrect'.

The correctness of the solutions was assessed by two independent consultants from another research institute. They were provided with task specifications, correct solutions and guidelines for assessment. They were not involved in the experiment or teaching of the participants. All solutions were compiled, executed and tested thoroughly for functionality. The source code was also inspected manually. To avoid inconsistencies, the solutions were analysed by both consultants independently and differences were resolved through discussion. The consultants also provided a detailed list of errors for each participant. A web-based tool (Arisholm and Sjøberg, 2004) was used to grade the solutions. This took about 80 working hours.

4. Results

We first analysed the raw data to see whether there were any errors due to technical problems or misunderstandings during the experiment. It transpired that one participant had major technical problems (with the network connection) and was not able to follow the experimental procedure. His data was excluded from the analysis. His background was average. This left 38 participants for the analysis.

We then identified the strategies applied by the participants as described in Section 3.6.1. Twenty-two participants applied the systematic strategy and 16 participants applied the as-needed strategy. We have analysed the background data for the participants to check if there were some significant differences between these two groups. Eight of 38 participants (21%) had more knowledge than average. Among the participants who applied the systematic strategy were four of them (18% of all the participants who applied the systematic strategy). Among the participants who applied the as-needed strategy were another four of them (25% of all the participants who applied the as-needed strategy). Thirteen of 22 participants (59%) who applied the systematic strategy and 8 of 16 participants (50%) who applied the as-needed strategy were from the same institution. We thus believe that there were no significant differences between these two groups.

Section 4.1 presents results on performance. Section 4.2 describes difficulties the participants experienced while performing maintenance tasks. Section 4.3 describes interactions between these difficulties and the strategy applied by the participants. Section 4.4 describes the participants' choice of strategy. Section 4.5 summarises and discusses the results.

4.1. Correctness and solution time

Table 4 presents the descriptive statistics for Task 1, Task 2 and both tasks related to the research question RQ1. The column *Correct* shows the percentage of the participants that delivered a correct solution. The columns from *Mean* to *Q3* present the descriptive statistics of the solution times in minutes. The *Total* row shows that 74% of the participants delivered correct solutions for Task 1, 82% for Task 2 and 63% for both tasks. Furthermore, the median time that the participants spent on Task

1 was 47 minutes; the median time for Task 2 was 69 minutes; the median time for both tasks was 109 minutes. The differences between the strategies are highlighted in Figure 1.

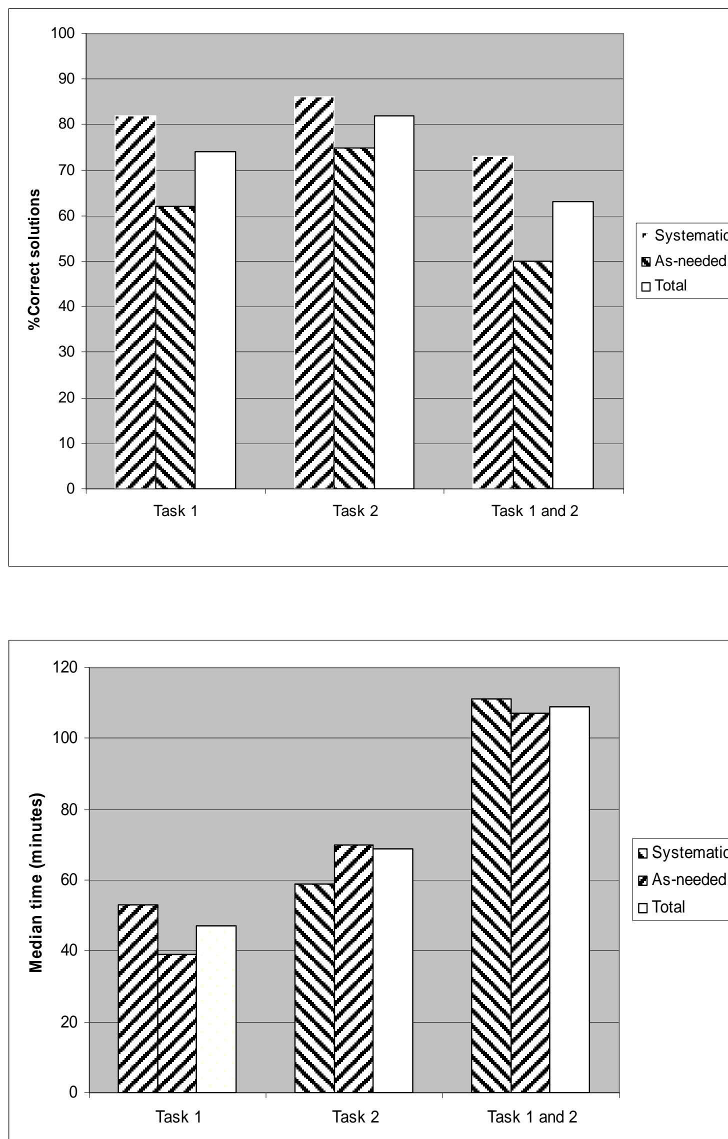


Fig. 1. Percentage of correct solutions and median of time participants spent to solve the tasks

The participants were more likely to produce correct solutions when applying the systematic strategy (82% versus 62% for Task 1; 86% versus 75% for Task 2; 73% versus 50% for both tasks).

Due to our definition of the systematic strategy, the participants who applied it were expected to spend a longer time on Task 1 than those who applied the as-needed strategy. This turned out to be correct, with users of the systematic strategy spending a median time of 53 minutes to complete Task 1 and those of the as-needed strategy spending a median time of 39 minutes. For the second, more complex, task the situation was reversed. Those participants who applied the systematic strategy spent a shorter time than did those who applied the as-needed strategy (59 versus 70 minutes). It would seem that the time participants spent on acquiring an overview of the systems at the beginning in Task 1 paid off later in Task 2. However, all participants spent approximately the same time to complete both tasks (112 minutes for the systematic strategy; 107 minutes for the as-needed strategy).

Table 4
Descriptive statistics of correctness and time

			Correct	Time (minutes)						
	Strategy	N	(percent)	Mean	Median	StD	Min	Max	Q1	Q3
Task1	Systematic	22	82%	55	53	27	27	147	39	57
	As-needed	16	62%	46	39	27	18	129	27	62
	Total	38	74%	52	47	27	18	147	31	57
Task2	Systematic	22	86%	64	59	20	20	99	52	82
	As-needed	16	75%	83	70	33	47	179	66	93
	Total	38	82%	72	69	27	20	179	53	83
Tasks 1 and 2	Systematic	22	73%	119	112	28	76	197	104	134
	As-needed	16	50%	129	107	50	77	245	93	163
	Total	38	63%	123	109	39	76	245	96	141

Fig. 1. Percentage of correct solutions and median of time participants spent to solve the tasks

We also analysed user action logs collected by GRUMPS to find the most visited classes and to check whether there were any differences between the participants who applied the systematic strategy and those who applied the as-needed strategy. A solution prepared in advance by the authors was used for this comparison.

The most visited classes for each task were, in fact, those that needed to be edited in order to complete the tasks successfully. Table 5 shows the time the participants spent in the ten most visited classes and the time they spent in the classes that needed to be edited. These times are given in minutes per participant. The time the participants spent on reading the documentation, and on compiling and running the application, is not included here. During the first task the participants that applied the systematic strategy spent a larger proportion of their time on the classes that required editing (44% of the mean time) than the participants that applied the as-needed strategy (35% of the mean time). For the second task this difference was smaller: 80% of the mean time for the systematic strategy; 77% of the mean time for the as-needed strategy. An overview of the ten most visited classes for each task is presented in Appendix B.

Table 5
Distribution of time spent in classes per strategies

Strategy		Task 1 Time	% of mean	Task2 Time	% of mean
Systematic	10 most visited classes	33	60%	55	86%
	Needed editing	24	44%	51	80%
As-needed	10 most visited classes	22	48%	72	87%
	Needed editing	16	35%	64	77%

As described in Section 3.2, the participants of the experiment worked in different conditions, namely concurrent think-aloud, immediate retrospective think-aloud, feedback-collection method and control silent condition. To check whether the performance of the participants is influenced by the data collection method instead of, or in addition to, their strategy, we calculated the same descriptive statistics for each of the groups (Appendix C).

We found no evidence of exclusive use of the as-needed strategy among four groups. Furthermore, the participants were more likely to produce correct solutions when applying the systematic strategy with the exception of the RTA group for Task 1 and CTA group for Task2. For both tasks the percentage of correct solutions was higher or same for all four groups (CTA: 85% for systematic versus 0% for as-needed; RTA: 50% for both strategies; FCM: 83% for systematic versus 66% for as-needed; CS 66% for systematic versus 50% for as needed).

Whereas all the participants spent approximately the same time to complete both tasks (112 minutes for systematic; 107 minutes for as-needed), there were differences among four groups. The participants who worked in CTA condition and applied systematic strategy spent shorter time to solve both tasks than those who applied as-needed strategy (CTA: 108 minutes for systematic and 127 minutes for as needed). The participants who worked in RTA condition spent the same time to solve both tasks (127 minutes for both strategies) and the participants who worked in FCM and CS condition spent longer time to solve both tasks when applying the systematic

strategy (FCM: 113 minutes for systematic; 94 minutes for as-needed; CS: 120 minutes for systematic; 108 minutes for as-needed).

Based on the above described results, we conclude that the results on distributions of the strategies among participants and correctness indicated the same trends for all treatments. Trends regarding time were different for different treatments and we thus cannot draw any conclusions on time.

4.2. Difficulties in program comprehension

The participants experienced different difficulties while conducting change tasks. We first give an overview of the difficulties identified as a result of examining the participants' solutions, verbal protocols and the time spent by the participants in each class (Section 4.2.1). We then describe these difficulties in greater detail (Sections 4.2.2 and 4.2.3).

4.2.1. Overview of the difficulties

To identify the difficulties participants had, we first analysed reports made by the consultants and made a detailed list of errors for each participant. We then extended this list with the difficulties we found in the verbal protocols within the *comprehension and problems* category (category 3.3 in Appendix A).

The difficulties were then categorised within our refinement of the model of Von Mayrhauser and Vans (1995). Von Mayrhauser and Vans describe two types of knowledge: (i) general knowledge, which is independent of the specific software application that the programmers are trying to understand, and (ii) software-specific knowledge, which represents their level of understanding of the software application. They suggest that difficulties and errors in comprehension arise from a lack of either general, or specific knowledge, or both. Among the difficulties caused by the lack of general knowledge, we identified the following sub-categories: difficulties concerning general knowledge of graphical user interface (GUI), reuse of methods, and programming environment. Among the difficulties caused by the lack of the specific knowledge, we identified the following three sub-categories: difficulties concerning program logic, the given implementation of GUI, object-oriented comprehension and programming and testing procedure. Some of the difficulties resulted in delivering solutions with faults, whereas the others slowed the participants down but were eventually surmounted. If a difficulty belongs to the second group, we stated it explicitly in the description given below.

Table 6 gives an overview of the difficulties. A difficulty is presented only once per participant per task.

Table 6
Number and type of difficulties per tasks

	Task1	Task2
1 General		
1.1 General GUI knowledge		
1.1.1 Forgot to expand the window		11
1.1.2 No experience with GUI programming	1	1
1.2 Reuse of methods		1
1.3 Programming environment	1	
2 Specific		
2.1 Program logic	15	
2.2 GUI implementation		
2.2.1 Changing interface		7
2.2.2 Adding or removing GUI components	1	1
2.2.3 Removing label declarations	22	
2.3 OO comprehension and programming		
2.3.1 Overall program structure	1	
2.3.2 Impacts on classes		
2.3.2.1 Self-reported problems		3
2.3.2.2 Attributes in the wrong class		12
2.3.3 Impacts within classes		
2.3.3.1 Removing variables and methods	10	
2.3.3.2 Placing new attributes at the wrong place		2
2.3.4 Inherited functionality	3	7
2.4 Testing procedure	2	2

4.2.2. General difficulties

The participants reported problems related to the general knowledge of the graphical user interface (category 1.1). Eleven participants reported that they forgot to expand the window needed to allow extra fields in Task 2. They discovered this error when they tested their solutions and they managed to overcome this difficulty. This error might be due to their lack of experience with GUI programming. Two participants stated explicitly they had never done any interface programming before.

One participant copied and pasted the body of a method in order to reuse it and never subsequently conducted any changes of this method in Task 2 (category 1.2). It is possible that the participant was not sufficiently careful, but the action may also indicate a lack of knowledge of reuse in object-oriented programming.

One participant reported that he had problems with JBuilder, because he had no experience with this tool (category 1.3). From his solutions we have seen that he used this tool. However, the lack of the experience might have slowed him down.

4.2.3. Specific difficulties

A difficulty that was reported quite often (15 occurrences for Task 1) was related to an if-then-else statement and was presented in category 2.1. This statement implements a book search on title, author or ISBN. The participants removed either too much or too little from this statement, and as a result, the library application did not work properly. It might be that the participants interpreted this task as a pure text editing task (finding all ISBN occurrences and deleting them) and did not try to

understand the logic of the program. However, it is also possible that the participants felt time pressure and were not sufficiently careful.

The participants had difficulty in understanding the implementation of the graphical user interface of the library application (category 2.2). They made different errors when changing the interface, such as reading the wrong attribute into a textbox, and not initialising a text field after saving it (seven occurrences in Task2; category 2.2.1). Two participants forgot to add or remove different components of the GUI, like radio buttons and text fields (one occurrence in Task 1, one occurrence in Task 2; category 2.2.2). The most common error related to the user interface was a failure to remove all label declarations of ISBN in Task 1 (22 occurrences; category 2.2.3). It was clearly stated in the task description that all references to ISBN should be removed. However, leaving some of these ISBN declarations had no effect on the functionality of the application. Some participants explained their decision in the verbal protocols. This is illustrated by the following comment written on the feedback-collection screen:

> "I was wondering whether to go through all the code and remove everything that had to do with ISBN. I have asked about this. I will only remove what is visible in the GUI, or else it will be too much work!"

The participants had different difficulties in understanding the structure and functionality of the library application. The majority of these difficulties were related to the object-oriented structure of this application. One participant reported having problems in understanding the overall structure of the library application in Task 1 (category 2.3.1). Many participants had problems finding classes affected by a change (category 2.3.2). Three participants reported in verbal protocols that this was difficult in Task 2 (category 2.3.2.1). Twelve participants placed attributes in the wrong class in Task 2 (category 2.3.2.2). Identifying the effects of changes within a class was also difficult for the participants. Ten participants failed to remove all variables and methods affected by a change after identifying a class affected by the change in Task 1 (category 2.3.3.1). Two participants placed new attributes at the wrong place in Task 2 (category 2.3.3.2).

Some participants had problems with inheritance of the functionality. Classes in the library application that needed to be persistent had to inherit an abstract class called Persistent. The subclasses of the Persistent class had to implement the methods *read()* and *write()*, which reads/writes from/to a file. The failure to make data persistent occurred relatively often (category 2.3.4; three occurrences for Task 1, seven occurrences for Task 2).

The testing procedure was described in the documentation. The participants should test their library applications by inserting new records in the library database. They should check if these records were stored in the database by reading a text file. Some participants had problems with finding and removing these files (category 2.4; two occurrences in Task 1, two occurrences in Task 2). As this was described in the documentation, this might be because they did not read it carefully. The participants managed to overcome these difficulties.

4.3. Difficulties and strategies

Table 7 gives an overview of the difficulties per strategy. An 'S' means that the participants applied the systematic strategy, an 'A' means that the participants applied the as-needed strategy. One should be aware that the number of participants who applied the systematic strategy is larger than the number of participants who applied as-needed strategy (22 participants applied systematic strategy; 16 participants applied as-needed strategy). We thus also give a number of difficulties per participant. The numbers of difficulties per strategy are relatively small and the results should therefore be treated cautiously. A more detailed survey of the difficulties per strategy is given in Appendix D.

Table 7
Number and type of difficulties per task and strategy. Number of difficulties per participant is given in parentheses.

	Task 1		Task 2	
	S N=22	A N=16	S N=22	A N=16
1 General				
1.1 General GUI knowledge	1 (0.04)		6 (0.27)	6 (0.37)
1.2 Reuse of methods				1 (0.06)
1.3 Programming environment	1 (0.04)			
Total - General	2 (0.09)		6 (0.27)	7 (0.31)
2 Specific				
2.1 Program logic	8 (0.36)	7 (0.43)		
2.2 GUI implementation	12 (0.54)	11 (0.68)	3 (0.14)	5 (0.31)
2.3 OO comprehension and programming	9 (0.4)	5 (0.31)	11 (0.5)	13 (0.81)
2.4 Testing procedure		2 (0.12)		2 (0.12)
Total - Specific	29 (1.31)	25 (1.56)	14 (0.64)	20 (1.25)
Total	31 (1.4)	25 (1.56)	20 (0.9)	27 (1.68)

The total number of difficulties was slightly smaller for the systematic strategy (51 difficulties in total for systematic versus 52 for as-needed; 2.3 difficulties per participant for systematic versus 3.25 for as-needed). As the participants that applied the systematic strategy spent their time during Task 1 on getting an overview of the application, we expected that they would have a smaller number of difficulties related to the knowledge of the program structure in Task 2. The number of difficulties related to understanding of the GUI implementation and to the understanding of the object-oriented structure of the application was indeed slightly smaller for the systematic strategy. GUI-implementation related difficulties occurred less frequently for the systematic strategy than for the as-needed strategy in Task 2 (category 2.2; three occurrences for systematic versus five occurrences for as-needed; 0.14 difficulties per participant for systematic versus 0.31 difficulties per participant for as-needed). Difficulties related to object-oriented comprehension and programming also occurred less frequently for the systematic strategy than for the as-needed strategy (category 2.3; 11 occurrences for systematic versus 13 occurrences for as-needed; 0.5 difficulties per participant for systematic versus 0.81 difficulties per participant for as-needed).

It is interesting that in Task 1, the number of difficulties related to removing variables and methods was much larger for the participants who applied the

systematic strategy (Appendix D, category 2.3.3.1; eight occurrences for the systematic strategy versus two occurrences for the as-needed strategy). We have no explanation for this.

As explained above, the testing procedure was described in the documentation. Only the participants who applied the as-needed strategy and who, as a result, probably did not pay enough attention to the documentation, had difficulties with doing this. They reported that they had problems in finding out the text files with database records (two occurrences in Task 1, two occurrences in Task 2).

4.4. Choice of strategy

To understand the reasons for selecting a particular strategy we conducted group interviews with the participants. They were asked to describe how they were working and whether they worked differently from normal.

The participants who applied the systematic strategy stated that they first wanted to get an overview of the program by reading the documentation, reading the source code and running the program. Some of them started with reading the documentation (UML diagrams), while others began by reading the code or running the program. The following examples illustrate this.

> "I started with the UML diagrams. Looked at them to gain an overview of the classes and where I thought I should do changes. Then I went into the code, and looked there."

> "I ran the application a couple of times first to see how it worked. Then I looked at the UML diagrams to see what was what."

> "I have to admit that I didn't use the UML diagrams. I went straight to the code to see what I should find instead. I used them a little bit after a while, but at first I looked at the code and understood how the system was built with packages and stuff. Then I started the program and saw where the functions we should use were, and then I went in where I thought I would find them."

The participants were aware that this strategy was time consuming. They put it in this way:

> "I spent too much time looking at the UML diagrams. I spent an enormous amount of time trying to understand the relationships."

> "I like to work in a way where I try to understand as much as possible before I start, so I used quite a lot of time reading the documentation. I like to get an overview, what is the purpose of the program and things like that. In the beginning I read the documentation line by line, but I found out that I wouldn't get anything done if I continued in that manner. I looked at the UML diagrams, and I looked in the code to try to find the things from the diagrams, to understand the structure."

The participants explained that they were used to working in that way. This is illustrated by the following explanation:

> "I am used to gaining a broad understanding of an application before I start to work on it. That's why I was working in this way."

One participant explained that he read the documentation thoroughly because he thought that the tasks would be very difficult.

Several participants who applied the as-needed strategy explained that the functionality of JBuilder helped them to identify the code affected by a change without using the documentation. They also said that their choice of strategy was influenced by the properties of the given code (tidy, easy to understand, self-explaining class names) and the properties of the first task (subtractive change). One participant claimed that he started to change the code immediately because reading the documentation was too time-consuming. The following examples illustrate the participants' explanations.

> "I didn't use the diagrams much, because you have a good overview [of the places where ISBN occurs] to the left in the JBuilder window."

> "I thought that there would be ISBN almost everywhere. So I just used CTRL-F, or whatever it was to find ISBN. And I just searched through. In JBuilder I got a list of the ISBN occurrences, and I just commented them out. I didn't bother to look at the UML diagrams, because I thought this was a much easier way to do it."

> "I looked at the class diagram once. I had a look at it. Then I looked at the file names. They were self-explanatory, so because of that I thought OK, it has to work like this. And it did.

One participant regretted his strategy choice.

> "I didn't use the documentation much, but I probably should have. Especially on the last task."

The majority of the participants claimed that they worked as they usually did. However, a few participants said that they were influenced by the experimental situation (time pressure and the experimental procedure). For instance, one participant said:

> "I used the documentation because it was handed to me."

4.5. Summary and discussion

The participants of the experiment applied two strategies: systematic and as-needed, the first one being the more prominent. Twenty-two participants started by reading the documentation, reading the source code or running the application in order to get an

overview of the application and their initial strategy was classified as the systematic strategy. Sixteen participants started to conduct changes without trying to understand the system first and their initial strategy was classified as the as-needed strategy.

To provide a more realistic environment, we handed the documentation to the participants both electronically and on paper. Therefore, we used the user profiles (accessed files/documents/executions and time) and verbal protocols to identify strategies, instead of the access rates to files and functions as in (Koenemann and Robertson, 1991; Corritore and Wiedenbeck, 2001). Hence, we compare our results with the previous results in terms of the identified strategies instead of the access rates. Koenemann and Robertson (1991) reported that none of professional programmers that participated in their study followed systematic strategy when modifying 636 LOC large Pascal program. They spent a major part of their time searching for code segments relevant to the modification task and no time understanding the rest of the application. We find no support for the exclusive use of a highly localised strategy. Corritore and Wiedenbeck (2001) reported that both procedural and object-oriented programmers employed a wide breadth of comprehension when modifying given programs (783 LOC large C program for procedural programmers and 822 LOC large C++ program for object-oriented programmers). Our results accord with the results of Corritore and Wiedenbeck (2001), even though our application was much larger and the participants had less programming experience.

The participants in our experiment were more likely to perform better in terms of correctness when applying the systematic strategy (73% solved both tasks correctly for the systematic strategy, versus 50% for the as-needed strategy). Our results regarding correctness agree with those of Littman et al. (1986b), which showed that programmers who applied the systematic strategy were more successful in making changes to a small (200 LOC) procedural application.

Our analysis of difficulties revealed that the participants had difficulties due to a lack of general and specific knowledge. The major general difficulty was related to the general knowledge of GUI (category 1.1). Specific difficulties that occurred 10 or more times were related to the programming logic (category 2.1), identifying GUI components affected by a change (category 2.2.3), identifying classes affected by a change (category 2.3.2.2), identifying impacts of a change within a class (category 2.3.3.1) and using the inheritance of the functionality (category 2.3.4).

Tegarden and Sheetz (2001) reported that graduate students experienced the proper placement of functionality during the design of classes as a one of the difficulties of the object-oriented approach. Our results showed that the students not only found it to be difficult, but also failed to do it correctly. Détienne (1997) observed that novices had difficulties in using the inheritance of functionality during object-oriented design. Our results showed that using inheritance of the functionality was difficult during maintenance, for both additive and subtractive changes.

Littman et al. (1986b) explained the better performance of the programmers who applied the systematic strategy by citing the ability of this strategy to detect interactions of the code being modified with the code in other parts of the program. Our results showed that some difficulties related to the understanding of the GUI implementation and OO comprehension and programming occurred less frequently when the participants applied the systematic strategy. This partly provided evidential

support for the claim of (Littman *et al.*, 1986b). Nevertheless, these results should be treated cautiously, due to the small number of difficulties per type.

Previous studies in the procedural paradigm showed that the acquisition of strategy skills may play a significant role in the development of expertise (Davies, 1993). The participants in our experiment (advanced beginners) explained their choice of strategy by their preferences, the functionality of JBuilder, the properties of the given application and tasks. It seemed that some of them had the necessary tactical skills and were able to choose the strategy that is appropriate for the given application, tasks and tool.

5. Validity

This section discusses the most important threats to the validity of this study and what we have done to reduce them.

5.1. Measuring breadth of the comprehension

In order to make the experimental environments as close to everyday working practice as possible, we did not impose any restrictions regarding the use of documentation and other remedies. The documentation was given to the participants both electronically and on paper. The participants were also allowed to use books and the internet for help. However, we were thus not able to record exactly which documents they used and how much time they spent reading them. For example, if a longer period without activity was identified in log files, we assumed that the participants were reading the documentation on paper, but they might have been doing something else. To reduce this threat to validity, the data from log files was verified against the verbal reports. As some participants might be less talkative, or have forgotten to report usage of the paper documentation, we also used observers' notes.

5.2. Measuring time

The time measured as a difference between end_time and start_time might be affected by different breaks and disruptions during the experiment. To reduce this threat to validity, we reconciled the times recorded by the SESE tool against user action logs produced by GRUMPS and subtracted major nonproductive breaks from the task times. Reasons for breaks have been identified from the verbal protocols and observer notes.

5.3. Measuring correctness

The solutions were marked 'correct' if they worked and delivered all functionality described the given specification. Otherwise they were marked 'incorrect'. To ensure

the quality of the marking it was done by two independent consultants independently. The consultants were provided with all necessary experimental material and guidelines. All solutions were tested thoroughly for functionality and the source code was also inspected manually. The consultants compared their scores and resolved the differences through discussion. This, we believe, ensures the correctness of this measure.

5.4. Data collection methods

Due to the first goal of our experiment (evaluating usefulness of the feedback-collection method) the participants worked in four different conditions concurrent think-aloud, retrospective think-aloud, feedback-collection and control silent. Data collection methods could affect their choice of comprehension strategy.

The participants in our study were not aware that the study focuses, among other issues, on comprehension strategies. The participants were presented the tasks, the documentation and the application code in the same time. They could choose themselves whether or not they wanted to familiarise themselves with the program before proceeding with the performing the task. The majority of the participants said in the interviews that they worked as they are used to do, so we believe that the data collection method had no influence on the participants' choice of the strategy.

Data collection methods might also affect the performance of the participants. To reduce this threat to validity, we analysed the data on performance for each of the groups. The results indicated similar trends regarding correctness, but different trends regarding time. The next studies should eliminate this threat by using the same data collection methods for all of the participants.

5.5. Identifying difficulties

The data on difficulties experienced by the participants are partly qualitative and subjective. A detailed list of errors for each participant made by two independent consultants (difficulties that the participants did not overcome) was combined with the difficulties identified in the verbal protocols (both the difficulties that the participants did and did not overcome). These verbal protocols were coded by two independent coders to ensure correctness. However, one should be aware that there might be differences among participants. Some might be less talkative, or have forgotten to report their difficulties. Hence, some of the difficulties the participants had during the experiment that they managed to overcome might be missing from our list. To reduce this threat to validity, we also checked the time spent by the participants in each class. No new difficulties were identified.

5.6. Application and tasks

The library application we used consists of four packages, 26 classes with 3600 LOC in total. The size of the classes varied from 24 to 311 LOC with an average of 166

LOC and median of 189 LOC. According to the classification given by Von Mayrhauser and Vans (1995), the application can be considered to be a medium-size application. It is possible that comprehension strategies, their effects on participants' performance, and the difficulties of the participants would be different for larger application systems and for more complex tasks. It is also possible that the results would be different if the experiment had lasted longer, i.e. if the participants had been given more time to become familiar with the application. Our results are, therefore, limited to situations in which developers had to comprehend and maintain a medium-size system, previously unknown to them and developed by others.

5.7. Participants

All the participants in this experiment were third or fourth year students in computer science and can be considered as novices according to the classification of Von Mayrhauser and Vans (1995), or as advanced beginners according to the classification of Dreyfus and Dreyfus (1986). We might expect similar results for programmers with a similar background. However, the results should be treated with caution, due to the relatively small sample size of this study; 38 participants completed the main experiment successfully. The participants' performance might vary not only because their strategy choice, but also because of personal differences. To address this problem, we analysed background data on the participants to check whether there were significant differences in programming skills among the groups of participants that chose different strategies to comprehend the library application. No significant differences were found. This, we believe, reduced this threat to validity.

6. Conclusions and future work

This paper provides empirical evidence regarding the comprehension strategies used by programmers while conducting maintenance tasks on an object-oriented application, and the difficulties they had. Our results showed that advanced beginners applied both the systematic and as-needed strategy while comprehending a medium-size Java application. The participants who applied the systematic strategy were more successful in producing correct solutions. The results revealed the difficulties the participants had due to a lack of knowledge that is independent of the specific application (general knowledge) and a lack of knowledge of the specific application (specific knowledge). The major specific difficulties were related to the comprehension of the application structure and to the inheritance of the functionality.

Two major difficulties (with more than 30 occurrences) were related to the understanding of GUI implementation and OO comprehension and programming. Our results indicated that these two groups of difficulties occurred less frequently in Task 2 when the participants applied the systematic strategy. The results of the interviews showed that the participants' choice of comprehension strategy was partly influenced by the functionality of the development tool and the properties of the given application and tasks.

Based on our results, we recommend that the programmers receive training in the advantages and disadvantages of different comprehension strategies for different maintenance tasks and applications. Furthermore, more attention should be paid in training to the areas in which the participants failed to apply their declarative knowledge of object-oriented concepts, such as inheritance of functionality.

Our goal was to ensure realism by using a professional development tool, providing participants with both on-line and paper documentation, and by allowing the participants to choose whether on not they wanted to familiarize themselves with the program before proceeding with the maintenance tasks. Nevertheless, there are several threats to the validity of our results, which threats we intend to address. We are going to replicate the experiment with different categories of professional developer. Further experiments also need to focus on larger applications and a variety of maintenance tasks.

The results regarding general and specific difficulties and their interactions with strategy choice are interesting in several ways. There are indications that the acquisition of application-specific knowledge can be improved by choosing the systematic strategy. However, it is unclear why this effect was not identified for all difficulties related to comprehension of the program structure. Furthermore, it is surprising that a relatively large number of difficulties arose due to relying on the development tool and avoiding comprehension of the program logic. We intend to further explore interactions among strategic, programming and tool knowledge, and the effects of these interactions on program comprehension.

Acknowledgements

This research was supported by the Comprehensive Object Oriented Learning project (COOL). Richard Thomas acknowledges support from the Simula Research Laboratory Guest Researcher programme and Amela Karahasanović acknowledges support from the University of Western Australia Gledden Visiting Senior Fellowship.

The authors are grateful to Unni Nyhamar Hinkel, Ray Welland and Chris Wright for valuable contributions to this paper. We thank Gunnar Carelius for his technical assistance; Per Thomas Jahr and Bent Østebø Johansen from the Norwegian Computing Centre for assessing the quality of the solutions; Kirsten Ribu for her contribution to the organisation and conducting of the experiment. We are grateful to Jo Hannay, Vigdis Kampenes By and Nils-Kristian Liborg for testing the experimental material and to Thorbjørn Hermansen for providing a Java program for data analysis. We acknowledge Phil Gray and Iain McLeod from the University of Glasgow for adapting the GRUMPS software to our needs. We thank the members of the COOL project Annita Fjuk, Arne-Kristian Groven, Christian Holmboe, Jens Kaasbøll, Anders Mørch and Terje Samuelsen for their contributions to the conducting of the experiment. We thank the students of Oslo University College and the University of Oslo for their participation in this experiment and Naci Akkøk, Kjetil Grønning and Eva Vihovde for motivating the students to participate.

References

Arisholm, E. and Sjøberg, D. 2004. Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software. *IEEE Transactions on Software Engineering*, 30(8), 521–534.

Arisholm, E., Sjøberg, D.I.K., Carelius, G. and Lindsjørn, Y. 2002. A Web-based Support Environment for Software Engineering Experiments. *Nordic Journal of Computing*, 9, No. 4, 231–247.

Arisholm, E., Sjøberg, D.I.K. and Jørgensen, M. 2001. Assessing the Changeability of two Object-Oriented Design Alternatives – a Controlled Experiment. *Empirical Software Engineering*, 6, No. 3, 231–277.

Briand, L.C., Bunse, C., Daly, J.W. and Differding, C. 1997. An experimental comparison of the maintenability of object-oriented and structured design documents. *Empirical Software Engineering*, 2, 291–312.

Brooks, R.E. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 543–554.

Burkhardt, J.-M., Détienne, F. and Wiedenbeck, S. 1998. The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies. *6th International Workshop on Program Comprehension (IWPC'98)*, 82–89.

Burkhardt, J.M., Detienee, F. and Wiedenbeck, S. 2002. Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering*, 7, 115–156.

Chi, M.T. 1997. Quantifying Qualitative Analyses of Verbal Data: a Practical Guide. *The Journal of the Learning Sciences*, 6(3), 271–315.

Coleman, D., Ash, D., Lowther, B. and Oman, P. 1994. Using Metrics to Evaluate Software System Maintainability. *IEEE Computer, August 1994*, 44–49.

Corritore, C.L. and Wiedenbeck, S. 1999. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *Int. J. Human-Computer Studies*, 61–83.

Corritore, C.L. and Wiedenbeck, S. 2000. Direction and scope of comprehension-related activities by procedural and object-oriented programmers: An empirical study. *8th International Workshop on Program Comprehension*,

Corritore, C.L. and Wiedenbeck, S. 2001. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *Int. J. Human-Computer Studies*, 54, 1–23.

Dale, N. 2005a. SIGCSE members survey. http://www.cs.utexas.edu/users/ndale/ContentResults.html, accessed September 2006.

Dale, N. 2005b. Non SIGCSE members survey. http://www.cs.utexas.edu/users/ndale/ContentResults2.html, accessed September 2006.

Daly, J., Brooks, A., Miller, J., Roper, M. and Wood, M. 1996. Evaluating inheritance depth on the maintenability of object-oriented software. *Empirical Software Engineering*, 1, 31–48.

Davies, S.P. 1993. Int. J. Man-Machine Studies. 39, 237–267.

De Lucia, A., Fasolino, A.R. and Munro, M. 1996. Understanding Function Behaviours through Program Slicing. *4th Workshop on Program Comprehension (IWPC'96)*, 9–18.

Détienne, F. 1997. Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers*, 9, 47–72.

Détienne, F. 2002. *Software Design - Cognitive Aspects*. London: Springer.

Dreyfus, H.L. and Dreyfus, S.E. 1986. *Mind over Machine*. New York: The Free Press.

Dunsmore, A., Roper, M. and Wood, M. 2000. The role of comprehension in software inspection. *Journal of Systems and Software*, 52, 121–129.

Ericsson, K.A. and Simon, H.A. 1993. *Protocol Analysis: Verbal Reports as Data*. Cambridge, Massachusetts: A Bradford Book, MIT Press.

Eriksson, H.E. and Penker, M. (1998). "Case Study". In: *UML Toolkit*. (editors), New York, John Wiley & Sons, Inc.,

Henry, S. and Humphrey, M. 1993. Object-oriented vs. procedural programming languages: effectiveness in program maintenance. *Journal of Object-Oriented Programming*, 6(3), 41 – 49.

Holgeid, K.K., Krogstie, J. and Sjøberg, D.I.K. 2000. A Study of Development and Maintenance in Norway: Assessing the Efficiency of Information Systems Support Using Functional Maintenance. *Information and Software Technology*, 42, No. 10, 687–700.

Hughes, J. and Parkes, S. 2003. Trends in the use of verbal protocol analysis in software engineering research. *Behaviour & Information Technology*, 22(2), 127–140.

Johnson, R. and Foote, B. 1988. Designing reusable classes. *Journal of Object-Oriented Programming*, 1–2, 22–35.

Karahasanovic, A., Anda, B., Arisholm, E., Hove, S.E., Jørgensen, M., Sjøberg, D.I.K. and Welland, R. 2005. Collecting Feedback during Software Engineering Experiments. *Empirical Software Engineering*, 10, No. 2, 113–147.

Karahasanovic, A., Hinkel, U.N., Sjøberg, D.I.K. and Thomas, R. 2004. Feedback Collection versus Think-Aloud in Software Engineering Research: A Controlled Experiment. *submitted to Behaviour & IT, available as Technical Report 2004-8, Simula Research Laboratory*,

Khazaei, B. and Jackson, M. 2002. Is there any difference in novice comprehension of a small program written in the event-driven and object-oriented styles? *IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 19–26.

Ko, A.J. and Uttl, B. 2003. Individual differences in program comprehension strategies in unfamiliar programming systems. *11th IEEE International Workshop on Program Comprehension*, 175–184.

Koenemann, J. and Robertson, S.P. 1991. Expert problem solving strategies for program comprehension. *SIGCHI conference on Human factors in computing systems: Reaching through technology,* New Orleans, Louisiana, ACM Press New York, NY, USA, 125–130.

Lehman, M. and Belady, L.A. 1985. *Program Evolution – Processes of Software Change*. Academic Press, London.

Lientz, B.P. 1983. Issues in Software Maintenance. *Computing Surveys*, 15 (3), 271–278.

Littman, D.C., Pinto, J., Letovski, S. and Soloway, E. 1986a. Mental models and software maintenance. *First Workshop on Empirical Studies of Programmers,* Norwood, NJ, Ablex, 80–98.

Littman, D.C., Pinto, J., Letovski, S. and Soloway, E. (1986b). "Mental models and software maintenance". In: *Empirical Studies of Programmers*. Soloway, E. and Iyengar, S. (editors), Norwood, NJ, Ablex, 80–98.

Mayer, B. 1988. *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall.

Mosemann, R. and Wiedenbeck, S. 2001. Navigation and Comprehension of Programs by Novice Programmers. *IEEE 9th Int. Workshop on Program Comprehension (IWPC 2001)*, 79–88.

Nosek, J.T. and Prashant, P. 1990. Software Maintenance Management: Change in the Last Decade. *Journal of Software Maintenance Research and Practice*, 2, No. 3, 157–174.

O'Brien, M.P. and Buckley, J. 2001. Inference-based and Expectation-based Processing in Program Comprehension. *9th International Workshop on Program Comprehension, 2001. IWPC 2001*, 71–78.

Parkin, P. 2004. An exploratory study of code and document interaction during task-directed program comprehension. *Australian Software Engineering Conference*, 221–231.

Pennington, N. 1987a. Comprehension strategies in programming. *Empirical studies of programmers, Second Workshop,* Ablex Publishing Corporation, 100–113.

Pennington, N. 1987b. Stimulus structures and mental representations in experts comprehension of computer programs. *Cognitive Psychology*, 19, 295–341.

Pfleeger, S.L. 1987. *Software Engineering – The Production of Quality Software*. Macmillan.

Rosson, M.B. and Alpert, S.R. 1990. The cognitive consequences of object-oriented design. *Human-Computer Interaction*, 5, 345–379.

Sheetz, S.D. 2002. Identifying the difficulties of object-oriented development. *The Journal of Systems and Software*, 64, 23–36.

Shneiderman, B. and Mayer, R. 1979. Syntactic/semantic interactions in program behaviour: A model and experimental results. *International Journal of Computer and Information Sciences*, 8, No. 3, 219–238.

Soloway, E., Adelson, B. and Ehrlich, B. (1988). “Knowledge and Processes in the Comprehension of Computer Programs”. In: *The Nature of Expertise*. Chi, M. *et al.* (editors), 129–152.

Storey, M.A. 2005. Theories, Models and Tools in Program Comprehension: Past, Present and Future. *13th International Workshop on Program Comprehension*, 181–191.

Storey, M.A., Fracchia, F.D. and Muller, H.A. 1999. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44, 171–185.

Tegarden, D.P. and Sheetz, S.D. 2001. Cognitive activities in OO development. *Int. J. Human-Computer Studies*, 54, 779–798.

Thomas, R., Kennedy, G., Draper, S., Mancy, R., Crease, M., Evans, H. and Gray, P. 2003. Generic Usage Monitoring of Programming Students. *ASCILITE 2003 Conference, University of Adelaide, Australia,* Adelaide, Australia, ASCILITE, 715–719.

Torchiano, M. 2004. Empirical investigation of a non-intrusive approach to study comprehension cognitive models. *Eighth European Conference on Software Maintenance and Reengineering*, 184-192.

Upchurch, R. 2002. Code reading and program comprehension: annotated bibliography. http://www2.umassd.edu/SWPI/ProcessBibliography/bib-coderading2.html, accessed September 2006.

Van Someren, M.W., Barnard, Y.F. and Sandberg, J.A.C. 1994. *The thinking aloud method: A practical guide to model cognitive processes*. London: Academic Press.

Von Mayerhauser, A. and Vans, A.M. 1993. From Code Understanding Needs to Reverse Engineering Tool Capabilities. *Sixth Int. Workshop on Computer-Aided Software Engineering CASE'93,* Singapore, 230–239.

Von Mayerhauser, A. and Vans, A.M. 1995. Industrial Experience with an Integrated Comprehension Model. *Software Engineering Journal*, 171–182.

Von Mayrhauser, A. and Vans, A.M. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8), 44–55.

Von Mayrhauser, A. and Vans, A.M. 1996. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22 No. 6, 424–437.

Von Mayrhauser, A. and Vans, A.M. 1997. Hypothesis-driven understanding processes during corrective maintenance of large scale software. *Int. Conf. on Software Maintenance*, 12–20.

Wiedenbeck, S. and Ramalingam, V. 1999. Novice Comprehension of small programs written in the procedural and object-oriented styles. *Int. J. Human-Computer Studies*, 51, 71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S. and Corritore, C.L. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11, 255–282.

Zelkowitz, M.V. 1978. Perspectives on Software Engineering. *ACM Computing Surveys*, 10, No. 2, 197–216.

Appendix A

Table A.1. Coding schema. In practice, one segment may belong to more than one category. More details on the coding schema can be found in (Karahasanovic *et al.*, 2005).

	Code	Example
1	Experimental context	
1.1	Breaks and disruptions	"Coffee break."
1.2	Background knowledge	"I have never used JBuilder before."
1.3	Experimental material	"Should I delete the ISBN field or write it as a comment?"
1.4	Supporting tools	"Problems with SESE."
1.5	Physical environment	"I don't have enough space to work."
2	Participants' perceptions	
2.1	Stress	"I am tired and have problems to concentrate."
2.2	External disturbance	"I am being disturbed by my neighbour."
2.3	General reflections	"Everything fine."
3	Experimental conduct	
3.1	Task-performing actions	
3.1.1	Actions on code (read, edit, search,	"I removed all ISBN fields from the
3.1.2	compile)	Business Objects classes."
3.1.3	Reading the documentation	"I have started by reading the UML diagrams."
	Running the library application	"I'll run the library application to test my changes."
3.2	Planning, strategy and reflection	"If I had more time I would rewrite the class BorrowerInformation to be more general."
3.3	Comprehension and problems	"I have problems seeing the big picture."

Appendix B

Table B.1. Time the participants spent in a class. The time is given in minutes per participant. The classes that had to be altered in order to complete the task are marked by a '٧' sign in front of the class name.

Task	Needed editing	Class name	Time
1	٧	Title.java	5.0
	٧	FindTitleDialog.java	4.9
	٧	TitleInfoWindow.java	4.2
	٧	TitleFrame.java	3.6
	٧	UpdateTitleFrame.java	2.8
		StartClass.java	2.2
		Persistent.java	1.3
		ObjId.java	1.1
		LendItemFrame.java	1.1
		BrowseWindow.java	1.0
2	٧	BorrowerFrame.java	18.7
	٧	BorrowerInformation.java	12.2
	٧	UpdateBorrowerFrame.java	11.8
	٧	BorrowerInfoWindow.java	8.2
	٧	FindBorrowerDialog.java	5.4
		MainWindow.java	1.7
		Persistent.java	1.3
		Loan.java	0.9
		StartClass.java	0.7
		ObjId.java	0.5

Appendix C

Table C.1. Descriptive statistics of correctness and time per treatments. The analysis of the effects of data collection methods on correctness and solution time, and discussion of its practical importance are given in Karahasanovic *et al.* (2004).

				Correct	Time (minutes)				
		Strategy	N	(percent)	Mean	Median	StD	Min	Max
CTA	Task1	Systematic	7	85	67.1	56	37	40	147
		As-needed	2	0	53	53	18.4	40	66
		Total	9	67	64	56	33	40	147
	Task2	Systematic	7	100	59.7	54	18.9	35	87
		As-needed	2	50	124.5	124.5	77.1	70	179
		Total	9	89	74	56	43	35	179
	both	Systematic	7	85	117.9	108	40	77	197
		As-needed	2	0	127.5	127.5	24.7	110	145
		Total	9	67	138	111	51	95	245
RTA	Task1	Systematic	6	50	48.1	45.5	21.9	26	81
		As-needed	4	75	47.7	46	14.5	32	67
		Total	10	60	48	46	18	27	81
	Task2	Systematic	6	83	88	92	27.1	50	118
		As-needed	4	75	78.7	76	13	66	97
		Total	10	80	80	79	23	47	125
	both	Systematic	6	50	138.5	127	41.7	88	194
		As-needed	4	50	126.5	127	20.8	107	145
		Total	10	50	132	127	31	88	182
FCM	Task1	Systematic	6	100	55.7	54	24	29	100
		As-needed	6	66	42.7	28	42.5	18	129
		Total	12	83	49	35	24	18	129
	Task2	Systematic	6	83	54.5	52.5	22.9	20	90
		As-needed	6	83	67	68	11.1	48	83
		Total	12	83	62	66	19	20	90
	both	Systematic	6	83	110.1	113	23.3	76	143
		As-needed	6	66	112	94	56.3	77	226
		Total	12	75	110	96	38	76	212
CS	Task1	Systematic	3	100	51.7	52	4.5	47	56
		As-needed	4	75	39.5	38.5	12.7	25	56
		Total	7	86	45	47	12	25	56
	Task2	Systematic	3	66	68.3	73	10.8	56	76
		As-needed	4	75	80.5	73	22.3	63	113
		Total	7	71	75	71	18	56	113
	both	Systematic	3	66	120	120	12	108	132
		As-needed	4	50	123.5	108	40.4	95	183
		Total	7	63	120	114	25	95	169

Appendix D

Table D.1.
Number and type of difficulties per tasks and strategies. Number of difficulties per participant is given in parentheses.

	Task 1		Task 2	
	S N=22	A N=16	S N=22	A N=16
1 General				
1.1 General GUI knowledge				
1.1.1 Forgot to expand the window			5 (0.22)	6 (0.37)
1.1.2 No experience with GUI programming	1 (0.04)		1 (0.04)	
1.2 Reuse of methods				1 (0.06)
1.3 Programming environment	1 (0.04)			
2 Specific				
2.1 Program logic	8 (0.36)	7 (0.43)		
2.2 GUI implementation				
2.2.1 Changing interface			3 (0.14)	4 (0.25)
2.2.2 Adding or removing GUI components		1 (0.06)		1 (0.06)
2.2.3 Removing label declarations	12 (0.54)	10 (0.62)		
2.3 OO comprehension and programming				
2.3.1 Overall program structure		1 (0.06)		
2.3.2 Impacts on classes				
2.3.2.1 Self-reported problems			1 (0.04)	2 (0.12)
2.3.2.2 Attributes in the wrong class			7 (0.31)	5 (0.31)
2.3.3 Impacts within class				
2.3.3.1 Removing variables and methods	8 0.36)	2 (0.12)		
2.3.3.2 Placing new attributes at the wrong place				2 (0.12)
2.3.4 Inherited functionality	1 (0.04)	2 (0.12)	3 (0.14)	4 (0.25)
2.4 Testing procedure		2 (0.12)		2 (0.12)
Total	31 (1.4)	25 (1.56)	20 (0.9)	27 (1.68)