Information and Software Technology 51 (2009) 293–305

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

The effect of task order on the maintainability of object-oriented software

Alf Inge Wang [a,*], Erik Arisholm [b,c,1]

[a] Department of Computer and Information Science, Norwegian University of Science and Technology, Sem Sælandsvei 7-9, N-7491 Trondheim, Norway
[b] Simula Research Laboratory, PO Box 134, 1325 Lysaker, Norway
[c] Department of Informatics, University of Oslo, PO Box 1080, Blindern, N-0316 Oslo, Norway

ARTICLE INFO

Article history:
Received 6 October 2007
Received in revised form 5 March 2008
Accepted 11 March 2008
Available online 1 April 2008

Keywords:
Object-oriented design
Object-oriented programming
Maintainability
Maintenance planning
Software maintenance
Schedule and organizational issues

ABSTRACT

This paper presents results from a quasi-experiment that investigates how the sequence in which maintenance tasks are performed affects the time required to perform them and the functional correctness of the changes made. Specifically, the study compares how time required and correctness are affected by (1) starting with the easiest change task and progressively performing the more difficult tasks (Easy-First), versus (2) starting with the most difficult change task and progressively performing the easier tasks (Hard-First). In both cases, the experimental tasks were performed on two alternative types of design of a Java system to assess whether the choice of the design strategy moderates the effects of task order on effort and correctness.

The results show that the time spent on making the changes is not affected significantly by the task order of the maintenance tasks, regardless of the type of design. However, the correctness of the maintainability tasks is significantly higher when the task order of the change tasks is Easy-First compared to Hard-First, again regardless of design. A possible explanation for the results is that a steeper learning curve (Hard-First) causes the programmer to create software that is less maintainable overall.

1. Introduction

The effort required to make changes correctly to a software system depends on many factors. These factors include characteristics of the software system itself (e.g., code, design and architecture, documentation of the system, testability), the development environment and tools, the software engineering process used, and human skills and experience. For example, an empirical study by Jørgensen and Sjøberg [1] showed that the frequency of major unexpected problems is lower when tasks are performed by maintainers with a medium-level of experience than when they are performed by inexperienced maintainers. However, using maintainers with even greater experience did not result in any further reduction of unexpected problems. Further, the results of one of our previous empirical studies [2] showed that the effect of the design approach to a system on the time spent on, and correctness of, changes made depends not only on the design but also on the experience of the maintainers. In the study presented herein, we investigated how the breakdown and sequential ordering of maintenance tasks affects the time required to carry out change tasks and the resulting quality of the system. If we could find any indications that the way in which change tasks are ordered affects the maintainability of the system, the result would represent a high return on investment for software companies, because little effort is required to rearrange the task order.

In some cases, the priority of maintenance tasks is constrained by client priorities: *must have, good to have*, and *time permitting features/fixes* [3] or organisational goals [4]. In other cases, there are fewer constraints on how to break down and arrange the maintenance tasks, in which case one can choose freely among alternative strategies to prioritize or sequence the tasks. Thus, we wanted to assess the effects of ordering maintenance tasks with respect to *difficulty level*. We provide empirical evidence regarding two alternative strategies pertaining to the sequence of performing the change tasks: *Easy-First*, where the maintainers start with the easiest change task and progressively perform the more difficult tasks and *Hard-First*, where the maintainers start with the most difficult change task and progressively perform the easier tasks. Our decision to perform controlled experiments stems from the many confounding and uncontrollable factors that could blur the results in an industrial case study [5]. It is usually impossible to control for factors such as ability and learning/fatigue effects, and select specific tasks to assign to individuals in a real project setting. As a result, the threats to internal validity are such that it is difficult to establish a causal relationship between independent (e.g., Hard-First vs Easy-First) and dependent variables (e.g., time, correctness). We report results from two controlled experiments (which, taken together, form one quasi-experiment [henceforth,

* Corresponding author. Tel.: +47 73 59 44 85; fax: +47 73 59 44 66.
E-mail addresses: alfw@idi.ntnu.no (A.I. Wang), erika@simula.no (E. Arisholm).
[1] Tel.: +47 67 82 82 00; fax: +47 67 82 82 01.

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.03.005

the experiment]) that investigate whether the sequence of change tasks affects the correctness and the effort spent. Hence, the experiment attempted to assess two competing hypotheses:

(1) By starting with the easy maintenance tasks first, the learning curve will not be very steep, thus enabling the maintainers to obtain a progressively better overview of the software system before having to perform more difficult tasks. In this way, the maintainer is less likely to devise suboptimal solutions when performing the difficult tasks. This is closely related what is defined as the bottom-up strategy regarding program comprehension, in which programmers look for recognisable small patterns in the code and gradually increase their knowledge of the system [6].

(2) By starting with the difficult maintenance tasks first, the learning curve will be steep, as the programmer must obtain a more complete overview of the system before being able to perform changes. However, due to the better overview, the maintainer might be less likely to devise suboptimal task solutions. This is related to the top-down strategy regarding program comprehension, in which the programmer forms hypotheses and refinements of hypotheses about the system that are confirmed or refuted by items of the code itself [7].

The difficulty level of maintenance tasks is a nontrivial concept, and hard to predict precisely. In our experimental context, the difficulty level of the maintenance tasks was determined a priori by the authors, by considering the number and complexity of classes that would be affected by each change and an estimate of the time required to perform the tasks. The actual results of the experiment suggest that this approach was sufficiently accurate for the purpose of ranking the difficulty level of the tasks, as discussed further in Section 5.3.1. Such an approach might be amendable to industrial contexts as well, for example by performing a relatively informal and course-grained change impact analysis as a basis for the ranking.

The learning curve of a system also depends on how the system is structured. Hence, we also included two different design styles in the experiment: (i) a centralized control style design, in which one class contains most of the functionality and extra utility classes are used; and (ii) a delegated control style design, in which the functionality and data were assigned to classes according to the principles for delegating class responsibility advocated in [8]. These two design styles represent two extremes within object-oriented design. According to object-oriented design experts, a delegated control style is easier to understand and change than a centralized control style [9,10].

The software system to be maintained was a coffee vending machine [8]. The change tasks were relatively small (from 15 min to 2 h per task). In a case study of 957 maintenance or change requests in a company, Hatton found that above 75% these maintenance tasks lasted from 1 up to 5 h [11]. Admittedly, the experimental systems and tasks are small compared with industrial software systems and maintenance tasks. This is representative of a very common limitation in controlled software engineering experiments, but the impact of such limitations depends on the research question being asked and the extent to which the results are supported by theories that explain the underlying and more general mechanisms [12]. In this study, generalization of the results to larger systems and tasks can mainly be claimed based on the support of existing theories within program comprehension research (see Section 2). The results reported herein are supported by several of these theories (see Section 4.3). The impact on external validity are further discussed in Section 5.4.2, were we argue that the observed effects of task order on small systems might be *conservative* estimates of the effect that would be observed on larger systems.

The subjects were 3rd–5th year software engineering students who had no prior knowledge of the system being maintained. Given these experimental conditions, the results reported herein are not necessarily valid for experienced maintainers or maintainers who already have obtained a detailed understanding of the system that is to be maintained. However, we still believe that the scope of the study is highly relevant in an industrial context. This is because in our experience, it is common to assign new and inexperienced programmers to maintenance tasks, and unless careful consideration is given to the nature of the tasks assigned, such programmers may affect adversely the maintainability of the system. In addition, it has become more common to outsource the maintenance of a system to consultants who have no, or very little, prior knowledge of the system [13–15].

The remainder of this paper is organised as follows. Section 2 describes the theoretical background for the study. Section 3 describes the design of the experiment and states the hypotheses tested. Section 4 presents the results. Section 5 discusses what we consider to be the most important threats to validity. Section 6 concludes.

2. Maintainability of object-oriented software

The ISO 9126 [16] analysis of software quality has six components: functionality, reliability, usability, efficiency, maintainability, and portability. The ISO 9126 model defines maintainability as a *set of attributes that bear on the effort needed to make specified modifications*. Furthermore, maintainability is broken down into four subcharacteristics: analysability, changeability, stability and testability. However, these subcharacteristics are problematic in that they have not been defined operationally. Our experiment investigated how the process by which a software system is comprehended to perform changes affects maintainability; hence, it is principally the subcharacteristic analysability that is examined. Analysability is related to the process of understanding a system before making a change (program comprehension).

In addition to analysability, the experiment is related to changeability. Arisholm [17] views changeability as a two-dimensional characteristic: it pertains to both the *effort* expended on implementing changes, and the resulting *quality* of the changes. These (effort and resulting quality of changes) are also the quality characteristics we measured in the empirical study presented in this paper. There are several papers that describe studies that focus on making changes to a system ([18,19], and [20]). However, most of these studies focus on the *results* of changes to the software system and not the *process* of changing it, so they are not particularly relevant to our work. There are also papers that study how the relevant parts of the source code to be changed are found [21,22]. Our experiment only consider the effect of changing the sequence of change tasks in terms of required effort and resulting quality and does not report on how the individual maintainer makes the specific changes in the code.

In the following subsections, we elaborate upon the notion of analysability as it relates to software maintenance, program comprehension and we describes some empirical studies.

2.1. Analysability of object-oriented software

We define *analysability* as the degree to which a system's characteristics can be understood by the developer (by reading requirement, design and implementation documentation, and source code) to the extent that he can perform change tasks successfully. To be able to maintain and change a system efficiently (i.e., in a short-time) and correctly (i.e., with intended functionality and a minimum of side-effects) the maintainer must understand the sys-

tem well. This understanding can be achieved gradually, all at once, or somewhere in between. The process of coming to understand a system is also closely related to whether the change tasks will affect the whole system or only small parts of it. These issues are addressed below.

2.2. Analysability and program comprehension

Analysability is closely related to program comprehension. Program comprehension requires that the maintainer represent the software mentally [23,24]. A number of models have been proposed to describe the cognitive processes by which program comprehension may be achieved. Brook's model describes program comprehension as the initial developer's reconstruction of the knowledge domain [7]. In this model, program comprehension is achieved by recreating mappings from the real-world problems (problem domain) to the programming domain through several intermediate domains (e.g., inventories, accounting, mathematics, and programming languages). The intermediate domains are used to close the gap between the problem and programming domains. Letovsky has proposed a high-level comprehension model that consists of three main parts: a knowledge base, a mental model, and an assimilation process [25]. The maintainer constructs a mental representation by combining existing knowledge (e.g., programming expertise, domain knowledge) with the external representation of the software (documents and code) through a process of assimilation. Soloway, Adelson, and Ehrlich have proposed a model based on a top-down approach [26]. This model assumes that the code or type of code is familiar and that the code can be divided into subparts. The model suggests breaking down the code into familiar elements that will form the foundation for the internal representation of the system. Pennington's model is based on a bottom-up approach to program comprehension. Two mental representations are developed: a program model and a situation model [24]. The program model represents a control–flow abstraction of the code. The situation model represents the problem domain or the real-world. Shneiderman and Mayer have proposed another model, according to which the process of program comprehension transforms the knowledge of the program that is retained in short-term memory into internal semantic knowledge via a chunking process [27]. Mayrhauser and Vans have proposed a model they have named "integrated metamodel", which consists of four main components: the top-down model, situation model, program model and knowledge base [28]. The knowledge base is the foundation upon which the other three models are built, via a process of comprehension. This approach combines Pennington's bottom-up approach with Soloway, Adelson, and Ehrlich's top-down approach.

Another way of describing how programmers and maintainers achieve an understanding of a software system is to distinguish between opportunistic and systematic strategies. Littman et al. observed that these two strategies were used by programmers who were assigned the task of enhancing a personnel database program [29]. In the opportunistic (on-the-fly) approach, programmers focus only on the code related to the task, while in the systematic approach, they read the code, and analyse the control and data flows to gain a global understanding of the program. Littman et al. found that programmers who used the opportunistic approach only acquired information about the structure of the program, while programmers who used the systematic approach acquired, in addition, knowledge of how the components in the program interact when it is executed. Further, the study showed that programmers who used the opportunistic approach made more errors because of their lack of understanding of how the components in the program interact.

The various models of program comprehension may be categorized as top-down, bottom-up or a combination of the two. In the study presented in this paper, the Hard-First sequence for performing change tasks represents a top-down approach, while the Easy-First sequence represents a bottom-up approach. When using the top-down approach, the maintainer must acquire an overview of the system before he can make any changes to it. The maintainer will try to break the system as a whole into smaller and smaller parts until he understands the whole system. When using the bottom-up approach, the maintainer will start by concentrating on the portions of the code that need to be modified, which are easy to identify for simple tasks. As the maintainer realizes that more parts of the code need to be changed, he will gradually acquire an understanding of more and more, and possibly of all the system. This basic dichotomy regarding models of program comprehension is complicated a little when we consider the different design approaches. The delegated control style could enforce a top-down approach, because when making changes to the system, the maintainer must take into consideration the rather extensive interaction among the classes. For the centralized control style, most of the changes will be made to the main class, which contains most of the code.

Further, the Easy- and Hard-First task order can be related to opportunistic and systematic program strategies for comprehension, respectively. The opportunistic approach maps very well to the Easy-First task order, where only small parts of the system will be affected first and incrementally more parts will be affected. For the Hard-First task order, it is necessary to understand how the different parts of the system interact before any changes can be made.

The process of program comprehension depends on many factors, such as the characteristics of the program [24,30], individual differences between the programmers with respect to skill and experience [30,31], and the characteristics of the programming task itself [32]. For example, for simple tasks, the changes will probably only affect small portions of the code, but for more complex changes, the programmer must take into account the interaction between different parts of the system. Thus, in order to make more complex changes, it is important for the programmer to understand thoroughly the structure of the program and the interactions among its components. Results from studies by Pennington [24] show that for a task that requires recall and comprehension, the programmer will form an abstract view of the program that models control flow. To modify the program, the programmer will form a situation model that describe how the program will affect real-world objects. For example, the situation model describes the actual code "*numOfMonitors = numOfMonitors–sold*" as "reducing the inventory by the number of monitors sold".

2.3. Empirical studies on software maintenance

Several empirical studies have investigated program comprehension and software maintenance (see [33]). These studies were conducted to collect information to support the program comprehension models or to validate them, and range from observational (behavioral) studies to controlled experiments. Some of the models described in the previous paragraph (e.g., Pennington's model) are based on experiences of very small programs or parts of a program with 200 of lines of code or fewer. Corritore and Wiedenbeck describe an empirical study that investigated the mental representations of maintainers when working on software developed by expert procedural and object-oriented programmers [23]. In their study, the program being maintained consisted of about 800 lines of code, and hence was considered to be rather large. Empirical studies have also been conducted on real large-scale systems that contained over 40 K lines of code [28]. The program that was maintained in our experiment was a small system that contained about

400 lines of code. The focus of our experiment was different from that of the empirical studies cited in this paragraph. We sought to *investigate the resulting maintainability* of the system after changes have been made to it and *not how* the subjects comprehend the system per se.

3. Design of experiment

The results described in this paper are based on data from two controlled experiments, forming one quasi-experiment:

- *Controlled experiment 1*: The first of the two experiments evaluated the effect of a delegated (DC) versus a centralized (CC) control style design of a given system on maintainability [2]. In this controlled experiment, 99 junior, intermediate and senior Java consultants and 59 undergraduate and graduate students from the University of Oslo (UiO) participated. The subjects started with the easiest change task and progressively performed the more difficult tasks (Easy-First). Only the data from the 59 (3rd–5th year) students from the UiO that participated in that experiment (henceforth, the UiO experiment) are reused in this paper. The 99 java consultants were not included because the second experiment (see below) only contained (mostly 4th year) students, and we wanted to ensure that the level of education and experience of the subjects in the two experiments was similar. As discussed below, a pretest was used to further adjust for skill differences.
- *Controlled experiment 2*: The second controlled experiment was conducted with 66 (mostly 4th year) students at the Norwegian University of Science and Technology (NTNU), In the NTNU experiment the students followed the same experimental procedures and used the same materials as in the first experiment, except that they started with the most difficult task (randomly assigned to either the CC or DC design) and progressively performed the easier tasks (Hard-First). It can thus be classified as a *differentiated* replication [34] of the first experiment.

Thus, the two individual experiments formed a 2×2 factorial quasi-experiment [35] that had a total of 125 students as subjects, all of whom had comparable levels of education and experience. To further adjust for individual skill differences between the treatment groups, all subjects in both of the controlled experiments performed the same pretest programming task. The results of the pretest were then used in an analysis of covariance model of the effect of task order on maintainability [35]. This is a common approach for analysing quasi-experiments [35].

The subjects performed the maintenance tasks using professional development tools. To manage the logistics of this experiment, the subjects used the web-based Simula Experiment Support Environment (SESE) to download code and task descriptions.

Fig. 1 gives an overview of the experiment design that shows the differences between the UiO and NTNU experiments. The first three steps were the same for the UiO and NTNU experiments. However, for Tasks c1–c3, the subjects carried out four variants of the treatment with variation in two dimensions: *Task order* (Hard-First vs. Easy-First) and *control style* (DC vs. CC). The last step (Task c4) was the same for both experiments, but with two variations (DC and CC).

Careful consideration was made to ensure that the tasks given in fact had increasing (Easy-First) or decreasing (Hard-First) difficulty, as discussed further in Section 3.6.1. Note also that the subjects did not start with fresh code for every new change task, but instead built upon the system delivered after completion of the previous task. This arrangement was important as we wanted to assess the maintainability of the system by performing change task c4 after the three previous change tasks was performed.

3.1. Hypotheses

We now present the hypotheses tested in the experiment. The hypotheses aim to assess whether the task order (Easy-First vs. Hard-First) affects the dependent variables *duration* and *correctness* and whether the design (CC vs. DC) moderates the potential impact of the task order on duration and correctness, as depicted in Fig. 2.

The null-hypotheses of the experiment were as follows:

- $H0_1$–*The effect of task order on duration*. The time taken to perform change tasks is equal for the Easy-First and Hard-First task order.
- $H0_2$–*The moderating effect of design on duration*. The difference in the time taken to perform change tasks for Easy-First and Hard-First task order does not depend on design.
- $H0_3$–*The effect of task order on correctness*. The correctness of the maintained programs is equal for Easy-First and Hard-First task order.
- $H0_4$–*The moderating effect of design on correctness*. The difference in the correctness of the maintained programs for Easy-First and Hard-First task order does not depend on design.

Section 3.6 provides further details of the variables and statistical models used to formally specify and test the above hypotheses.

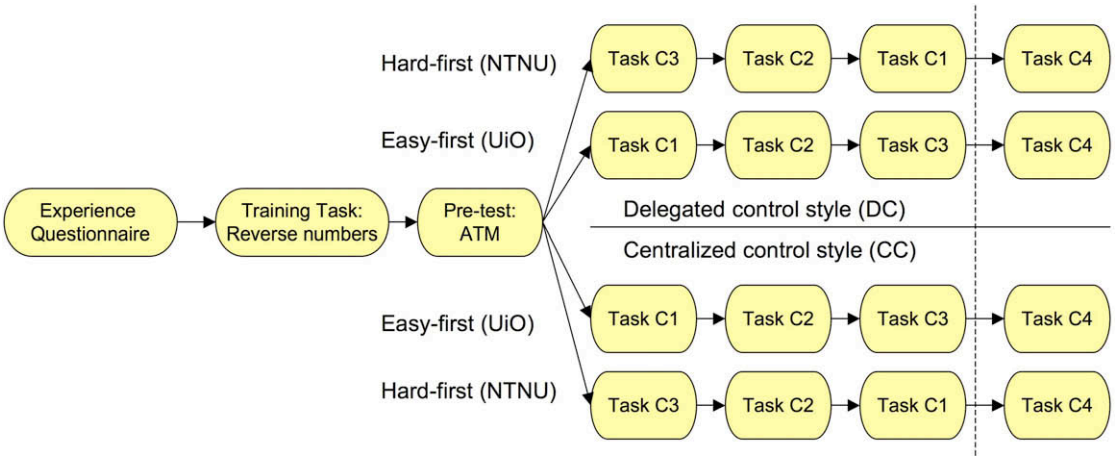

Fig. 1. An overview of the experiment design.

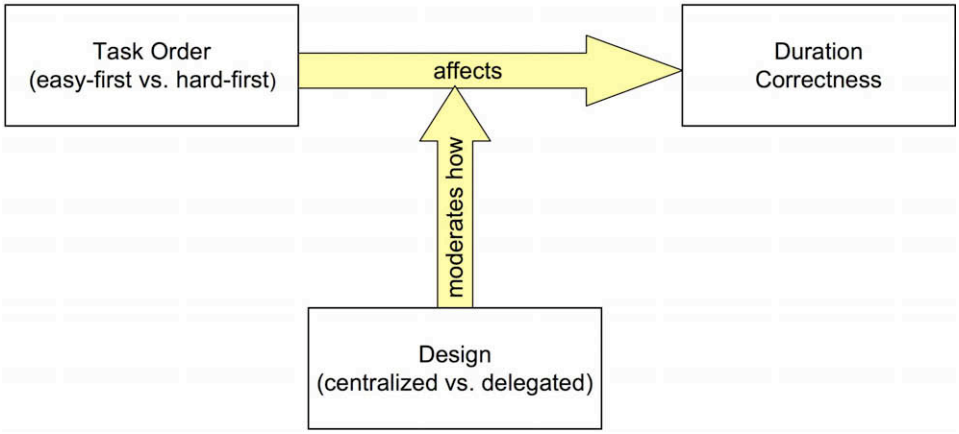

Fig. 2. Conceptual research model.

3.2. Design alternatives implemented in Java

Two alternative designs (CC and DC) of the coffee-machine were used as objects in the experiments, and were implemented in Java using similar coding styles, naming conventions, and amount of comments. For the centralized control style design one class contained most of the functionality and extra utility classes were used, while for the delegated control style design the functionality and data were assigned to classes according to the principles for delegating class responsibility advocated in [8]. All the names in the code used to identify variables and methods were long and descriptive. UML sequence diagrams were also provided, so that the subjects could obtain an overview of the main scenario for the two designs. The sequence diagrams are provided in [36].

3.3. Maintenance tasks

The maintenance tasks of the experiment consisted of six change tasks: a training task, a pretest task, and four (incremental) main experimental tasks (c1–c4). In the UiO experiment, the students performed the main tasks in the order c1, c2, c3 (Easy-First), and then c4. In the NTNU experiment, the students performed the tasks in the order c3, c2, c1 (Hard-First) and then c4. The c4 task occupied the same place in the sequence for both the UiO and NTNU experiments, as a benchmark task. For both UiO and NTNU, the code from the previous change task was used in the following change task.

To support the logistics of the experiments, the subjects used the web-based Simula Experiment Support Environment (SESE) [37] to complete a questionnaire on experience, download code and documents, upload task solutions, and answer task questions. The SESE tool was used for both experiments. The tool was also used to measure how much time the subjects spent on completing the change tasks. The experience questionnaire, detailed task descriptions, and change task questionnaire are provided in [36]. For each task in the experiment (see Fig. 1), the following steps were carried out:

- Download and unpack a compressed directory containing the Java code to be modified. This step was performed prior to the first maintainability task for the coffee-machine design change tasks (c1–c4), because these change tasks were related.
- Download task descriptions. (Each task description contained a test case that each subject used to test the solution.)
- Perform the task using a chosen Java development environment/tool.
- Pack the modified Java code and upload it to SESE.
- Complete a task questionnaire.

3.3.1. Training task

The training task required the subjects to change a small program so that it could read numbers from the keyboard and print them out in reverse order. The purpose of this task was to familiarize the subjects with the experimental procedures.

3.3.2. Pretest task

The pretest task asked the subjects to implement the same change on the same design for an automated bank teller machine. The change was to add transaction log functionality to a bank teller machine, and was not related to the coffee-machine designs. The purpose of this task was to provide a common baseline for assessing and comparing the programming skill level of the subjects. The pretest task had almost the same size and complexity as the subsequent three change tasks c1, c2, and c3 combined.

3.3.3. Main tasks

The change tasks for the coffee machine consisted of four changes:

- c1. Implement a coin-return button.
- c2. Introduce bouillon as a new drink choice.
- c3. Check whether all ingredients are available for the selected drink.
- c4. Make one's own drink by selecting from the available ingredients.

After completing tasks c1–c3, all the resulting systems would be functionally identical if the subjects had implemented them according to the provided specifications, regardless of whether the subjects had performed the tasks on the DC or CC design or in the Easy-First or Hard-First task order. Thus, task c4 could be used as a benchmark to measure the maintainability of the systems (as indicated by the time spent and the correctness of c4) after implementing c1–c3. Task c4 was more complex than tasks c1, c2, and c3. Note that the tasks c1–c3 are independent, but c4 depends on the previous change tasks.

3.4. Group assignment

In both the UiO and NTNU experiments, a randomized experimental design was used. Each subject was assigned randomly to one of two groups, CC and DC. In addition, the UiO students performed the maintainability tasks in the Easy-First task order, while the NTNU students performed them in the Hard-First task order

Table 1
Subject Allocation to treatment groups

	CC	DC	Total
Easy-First (UiO)	28	31	59
Hard-First (NTNU)	33	33	66

(see Fig. 1). The subjects in the CC group were assigned to the CC design and the subjects in the DC group were assigned to the DC design. Table 1 describes the distribution of number of subjects in the four groups used in the quasi-experiment. The reason for the uneven distribution for Easy-First (UiO) is that two of the subjects did not show up.

3.5. Execution and practical considerations

For the UiO experiment, graduate and undergraduate students in the Department of Informatics at UiO were contacted through e-mail and asked to participate. For the NTNU experiment, all students were recruited from the same software architecture course at the Department of Computer and Information Science at NTNU. In this course, most students are graduate students (4th year), but some are undergraduate (less than 10%). For the students at NTNU, the experiment was integrated as a voluntary exercise in the software architecture course. In both experiments, the students were paid about 1000 Norwegian Kroner (about $150) for participating in the experiment. The pay corresponds to eight hours wages as a course assistant and it was required that the students either complete the tasks of the experiment or work for up to eight hours on tasks in the experiment. The reason for paying the students was that in the UiO experiment Java consultants were paid to participate. To give the same conditions for all subjects participating, the students were also paid. At both universities, the results of the experiment were presented to the students when preliminary results from the analysis were available.

The subjects carried out the experiment in computer labs monitored by university staff, and the students were informed that they should only take breaks or have lunch between the change tasks and not during. The equipment and the data servers were also checked and monitored before and during the two controlled experiments, to make sure that the subjects did not lose any time because of problems with hardware or software. In addition, the training task in the two experiments (see Fig. 1) ensured that the subjects had the same familiarity with the experiment environment (SESE) and the programming environment. We observed no major disturbances during the two experiments.

3.6. Variables and model specifications

This section defines in more precise terms the variables of the experiment, how data was collected for these variables, and the models for analysis used to test the hypotheses.

3.6.1. Variables

Duration: Before starting on a task, the subjects wrote down the current time. When they had completed the task, they reported the total time (in minutes) that they spent on that task (the time was also recorded by the SESE tool). Two measures of duration were considered as dependent variables: (1) The elapsed time in minutes to complete change tasks $c1-c3$, and (2) the elapsed time in minutes to complete change task $c4$. Nonproductive time between the tasks was not included. For the duration measure to be meaningful, we considered the time spent only for subjects with correct solutions.

Correctness: For each task, test cases were devised to test the main scenario of the changed function. For each test run, the difference between the *expected* output of the test case and the *actual* output generated by each program was computed. The test case specifications are provided in [36]. The results of the functional tests and the actual code delivered were also thoroughly inspected by two hired third party Java experts to assess the degree of correctness further, in particular with regards to any regression faults that might not have been covered by the relatively simple, automated tests. They had no knowledge of the hypotheses of the experiment. For each task, and based on their inspection of the test case output and additional manual inspection of the code, a binary, functional correctness score was given. A task solution was assigned the value '1' if the task was implemented correctly and '0' if it contained serious logical errors. On the basis of the individual task correctness scores, two measures of correctness were considered as dependent variables: (1) The correctness of tasks c1–c3 ('1' if all three tasks were correct, '0' otherwise), and (2) the correctness of task c4.

TaskOrder: Describes whether the subjects performed the tasks starting with the *Easy-First* (c1, c2, c3, c4) or *Hard-First* (c3, c2, c1, c4) task order. The difficulty level of the maintenance tasks was determined a priori by the authors, by considering the number and complexity of classes that would be affected by each change and an estimate of the time required to perform the tasks. The actual results of the experiment suggest that this approach was sufficiently accurate for the purpose of ranking the difficulty level of the tasks (see Section 5.3.1).

Design: Describes the two alternative Java implementations of the coffee machine; centralized (*CC*) or delegated (*DC*).

Pretest duration (pre_dur): *The time taken (in minutes) to complete the pretest task (t1).* The individual pretest result was used as a covariate that modelled the variation in the dependent variables that could be explained by individual skill differences. Such an approach is known as Analysis of Covariance (ANCOVA), and is commonly used to adjust for differences between groups in quasi-experiments [35]. In our experiment, differences could be expected due to the fact that the experiment was conducted in two phases (UiO and NTNU) with two distinct samples of students. Furthermore, for the analysis of duration, we removed subjects with incorrect solutions, and this could also introduce differences between the groups that would be adjusted for with the pretest.

3.6.2. Model specifications

A generalized linear model (GLM) approach [38] was used to perform an ANCOVA to test the hypotheses specified in Section 3.1. The GENMOD procedure provided in the statistical software package SAS was used to fit the models. A justification for the specifications of the model follows.

Since this experiment was a quasi-experiment, the models needed to account for differences between the groups due to a lack of random assignment to the two task order treatments. The pretest measure *pre_dur* was used to specify ANCOVA models that adjust the observed responses for the effect of the covariate, as recommended in [35]. The covariate was log-transformed to reduce the potential negative effect that outliers can have on the model fit, among other things.

Furthermore, the duration and correctness data was not normally distributed, which also affected the model specifications. GLM is the preferred approach to analysing experiments with non-normal data [39]. In GLMs, one specifies the distribution of the response y, and a link function g. The link function defines the scale on which the effects of the explanatory variables are assumed to combine additively. The time data was modelled by specifying a *Gamma* distribution and the log link function. The *Gamma* distribution is suitable for observations that take only positive values and

Table 2
Complete model specifications

Model	Response	Distrib.	Link	Model term	Primary use of model term
(1)	Duration	Gamma	Log	Log(pre_dur)	Covariate to adjust for individual skill differences
				TaskOrder	Test $H0_1$ (duration main effect)
				Design	Models the effect of design on duration
				TaskOrder × Design	Test $H0_2$
(2)	Correctness	Binomial	Logit	Log(pre_dur)	Covariate to adjust for individual skill differences
				TaskOrder	Test $H0_3$ (correctness main effect)
				Design	Models the effect of design on correctness
				TaskOrder × Design	Test $H0_4$

are skewed to the right, which is the case for time data that has zero as a lower limit and no clear upper limit (though it cannot be longer than eight hours in this experiment). An alternative approach would be to simply log-transform the variable, by computing the log of each response log(y) as the dependent variable, and using a log-linear model to analyse the data on the assumption that log(y) would be approximately normally distributed. However, unlike such an approach, GLM takes advantage of the natural distribution of the response y, in our case *Gamma* for the time data. Furthermore, the expected mean $\mu = E(y)$, rather than the response y, is transformed to achieve linearity. As elaborated upon in [38,39], these properties of GLM have many theoretical and practical advantages over transformation-based approaches.

The correctness measure was fitted by specifying a *binomial* distribution and the *logit* link function in the GENMOD procedure. This special case of GLM is also known as a logistic regression model, and is a common choice for modelling binary responses.

Table 2 specifies the models. Given that the underlying assumptions of the models are not violated,[2] the presence of a significant model term corresponds to rejecting the related null-hypothesis. The following terms were used to test the hypotheses:

- The *TaskOrder* variable models the main effect of the *Easy-First* versus *Hard-First* task order on duration and correctness (to test hypotheses $H0_1$ and $H0_3$).
- The *design* variable models the main effect of the control style *DC* versus *CC* on *duration* and *correctness*, as an indicator of *system complexity*. The interaction term between *TaskOrder* and *design*, *TaskOrder* × *design*, models the moderating effect of the design on the effect of task order (to test hypotheses $H0_2$ and $H0_4$).
- The log-transformed covariate *Log(pre_dur)* adjusts for individual skill differences.

4. Results

This section presents the results from our quasi-experiment.

4.1. Descriptive statistics

The descriptive statistics of the experiment are shown in Table 3. The correctness and duration data are reported, both accumulated over the first three tasks (c1–c3) and for the final benchmark task (c4). The *Total N* shows the number of subjects assigned to the given treatment combination (e.g., CC and Easy-First). The *Correct N* column shows how many subjects actually solved the given task(s) correctly, and the *Correct %* column shows the proportion of subjects with correct solutions. As already discussed, duration is only reported for subjects with correct solutions on the given task(s), so the descriptive statistics for duration (*Mean, Std, Min, Q1, Med, Q3, Max*) are based on the *Correct N* number of observations. All tasks (c1–c3) are considered to be correct if all subtasks (c1, c2, and c3) are correct.

The main results (Easy vs. Hard-First for CC and DC for c1–c3) are visualized in Fig. 3. Duration is shown on the left *Y*-axis, while the percentage of correct solutions is shown to the right. There are some indications that there are interactions between control style and task order that affect duration and correctness. For example, for the CC design, correctness is better when starting with the easiest task first. In contrast, for the DC design, correctness is better when starting with the most difficult task first. The interaction effects are reversed when considering duration. For the CC design, the mean total duration required to perform all three tasks correctly is longer when starting with the easiest task first. In contrast, for the DC design, duration is shorter when starting with the most difficult task first.

When considering duration and correctness only for the final benchmark task (c4), the picture changes (see Fig. 4). Regardless of task order, more subjects who worked on the CC design seem to have correct solutions. Furthermore, the subjects who were assigned to the Hard-First task order have fewer correct solutions than those who started with the easy task first. However, there are no apparent interaction effects.

4.2. Hypothesis tests

The results of the hypotheses regarding correctness and duration for the first three tasks (c1–c3) are shown in Tables 4 and 5, respectively. The results suggest that TaskOrder does not have a significant impact on correctness or duration. There is some support for the hypothesis that control style (design) affects correctness ($p = 0.07$). There are no significant interaction effects between design and TaskOrder.

On the basis of the ANCOVA models, we also calculated the adjusted, least square means [36] for each model term, to assess and visualize the effect sizes for the two approaches to design (CC and DC) and TaskOrder (Easy-First and Hard-First) after adjusting for individual differences as indicated by the pretest. These estimates might be more reliable than the descriptive statistics, because they adjust for group differences. Because the model for the dependent variable duration used the *log* link function, the least square means estimates produced by the GENMOD procedure were first transformed back to the original time scale (in minutes) by taking the exponential of the adjusted least square means estimates. Similarly, the least square means of the logit, i.e., $\mu = \log(p/(1-p))$, was transformed back to the expected probability of having a correct solution ($p = \exp(\mu)/(\exp(\mu)+1)$. The results (including 95% confidence intervals) are shown in Tables 6 and 7 for correctness and duration, respectively. Table 7 shows that the most noticeable difference in estimates for correctness is a 16% difference for design (CC vs. DC) as expected, while the difference in estimates for the effect of task order is only 2%. Further, the estimates for duration in Table 7 show very small variations (a maximum of 5%).

[2] An empirical assessment of the model assumptions are provided in Section 5.1.

Table 3
Descriptive statistics of correctness and duration

Treatment variation				Correctness		Duration (in minutes)						
Design	Task order	Tot *N*	Task#	*N*	%	Mean	Std	Min	Q1	Med	Q3	Max
CC	Easy	28	c1–c3	20	71%	71	20	35	57	74	85	105
	First		c4	18	64%	78	31	25	60	70	98	142
	Hard	33	c1–c3	22	67%	63	28	18	47	59	78	128
	First		c4	8	24%	68	23	39	48	68	86	99
DC	Easy	31	c1–c3	15	48%	60	19	23	45	63	77	85
	First		c4	8	26%	96	39	37	72	88	125	160
	Hard	33	c1-c3	19	58%	68	33	30	46	59	81	146
	First		c4	4	12%	63	27	27	43	66	83	92

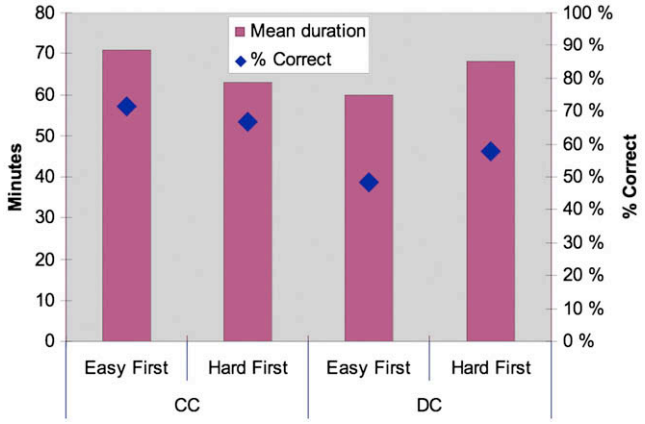

Fig. 3. The effects of design and task order on duration and correctness (tasks c1–c3).

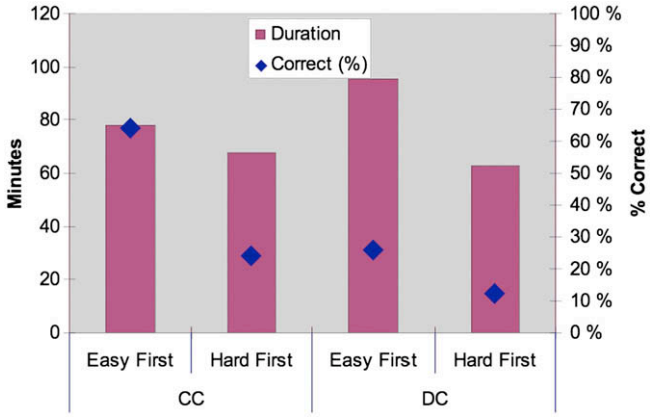

Fig. 4. The effects of design and task order on duration and correctness (tasks c4).

Table 4
ANCOVA results regarding correctness (tasks c1–c3)

Source	DF	Chi-square	Pr > ChiSq
pre_LogDur	1	8.71	0.0032
Design	1	3.09	0.0785
TaskOrder	1	0.09	0.7683
Design*TaskOrder	1	0.71	0.4003

When only considering the final task (c4) that we used as a benchmark, the results suggest that there are significant effects of both design ($p = 0.0022$) and TaskOrder ($p = 0.0001$) on correctness (Table 8). There are no significant interaction effects between design and TaskOrder ($p = 0.2839$). For duration there are no significant effects (Table 9). The corresponding effect size estimates are given in Tables 10 and 11 for correctness and duration, respectively. The least square means estimates for both main effects and interactions are visualized in Fig. 5.

We used change task c4 as a benchmark for testing the maintainability of the system. Hence, the main results of the experiment

Table 5
ANCOVA results regarding duration (tasks c1–c3)

Source	DF	Chi-square	Pr > ChiSq
pre_LogDur	1	26.24	.0001
Design	1	0.05	0.8147
TaskOrder	1	0.15	0.6977
Design*TaskOrder	1	0.45	0.5025

Table 6
Least squares means estimates for correctness (tasks c1–c3)

Effect	Design	TaskOrder	Estimate (%)	Lower 95% CL (%)	Upper 95% CL (%)
Design	CC		70	57	80
	DC		54	41	66
TaskOrder		Easy-First	63	50	75
		Hard-First	61	48	72
Design*TaskOrder	CC	Easy-First	74	55	87
	CC	Hard-First	65	47	79
	DC	Easy-First	51	33	69
	DC	Hard-First	56	39	73

Table 7
Least squares means estimates for duration (tasks c1–c3)

Effect	Design	TaskOrder	Estimate	Lower 95% CL	Upper 95% CL
Design	CC		64	58	71
	DC		63	57	71
TaskOrder		Easy-First	63	57	70
		Hard-First	65	59	72
Design*TaskOrder	CC	Easy-First	65	56	75
	CC	Hard-First	64	56	73
	DC	Easy-First	61	52	72
	DC	Hard-First	66	57	76

Table 8
ANCOVA results regarding correctness (tasks c4)

Source	DF	Chi-square	Pr > ChiSq
pre_LogDur	1	13.26	0.0003
Design	1	9.41	0.0022
TaskOrder	1	14.97	0.0001
Design*TaskOrder	1	1.15	0.2839

Table 9
ANCOVA results regarding duration (tasks c4)

Source	DF	Chi-square	Pr > ChiSq
pre_LogDur	1	6.20	0.0128
Design	1	1.09	0.2972
TaskOrder	1	0.76	0.3846
Design*TaskOrder	1	1.66	0.1973

Table 10
Least squares means estimates for correctness (tasks c4)

Effect	Design	TaskOrder	Estimate (%)	Lower 95% CL (%)	Upper 95% CL (%)
Design	CC		41	28	56
	DC		15	8	27
TaskOrder		Easy-First	46	32	61
		Hard-First	12	6	23
Design*TaskOrder	CC	Easy-First	69	49	84
	CC	Hard-First	18	8	35
	DC	Easy-First	25	12	45
	DC	Hard-First	8	3	22

Table 11
Least squares means estimates for duration (tasks c4)

Effect	Design	TaskOrder	Estimate	Lower 95% CL	Upper 95% CL
Design	CC		72	62	84
	DC		83	67	104
TaskOrder		Easy-First	83	71	96
		Hard-First	73	58	92
Design*TaskOrder	CC	Easy-First	70	59	84
	CC	Hard-First	74	57	96
	DC	Easy-First	97	76	124
	DC	Hard-First	72	50	103

concern the tests of the hypotheses for c4. We summarize the results of the hypothesis tests as follows:

- $H0_1$–*The effect of task order on duration*. The time on performing change tasks is equal for Easy-First and Hard-First task order: *Accepted*.
- $H0_2$–*The moderating effect of design on duration*. The difference in time taken to perform change tasks for Easy-First and Hard-First task order does not depend on design: *Accepted*.
- $H0_3$–*Effect of task order on correctness*. The correctness of the maintained programs is equal for Easy-First and Hard-First task order: *Rejected*.
- $H0_4$–*Moderating effect of design on correctness*. The difference in the correctness of the maintained programs for Easy-First and Hard-First task order does not depend on design: *Accepted*.

4.3. Summary of results

The most interesting results seem to pertain to correctness for the final task c4, as depicted in Fig. 5. We can see that the average proportion of correct solutions (aggregated across both task orders) was significantly higher for the CC design (41% correct) than for the DC design (15% correct). Moreover, the proportion of correct solutions was significantly higher when the task order was Easy-First (46%) as opposed to Hard-First (12%) (aggregated across both design approaches). If we consider the effects of task order on correctness for the two design approaches separately, we can see that the tendency was the same for both designs, but the effect was stronger for the CC design than for the DC design. For the CC design, subjects using the Easy-First task order produced significantly more correct solutions (69%) than did subjects using the Hard-First task order (18%). For the DC design the effect is less dominant, but here also, subjects using the Easy-First task order produced a higher proportion of correct solutions (25%) than those using the Hard-First task order (8%).

Interpretation of these results requires care. If we first consider the design approach's effect on correctness, a previous experiment of ours [2] showed that it is easier for programmers with limited experience (students, junior, and intermediate consultants) to maintain systems that use the CC approach than it is for them to maintain systems that use the DC approach. That experiment also showed that experienced programmers (senior consultants) can maintain systems that use the DC approach more efficiently than they can maintain systems that use the CC approach. This indicates that the DC approach requires a certain level of skill and experience to benefit from it. It is likely that the increased requirements with regards to skills and experience are due to delocalized plans [26], where the functionality is delegated across different objects in the system, which makes it more difficult to comprehend. These results are confirmed by the results reported in this paper.

One possible reason for the effects of task order is that by maintaining a system using an Easy-First task order, the learning curve will not be so steep. This means that the maintainer will learn the system in several increments, through which the maintainer's

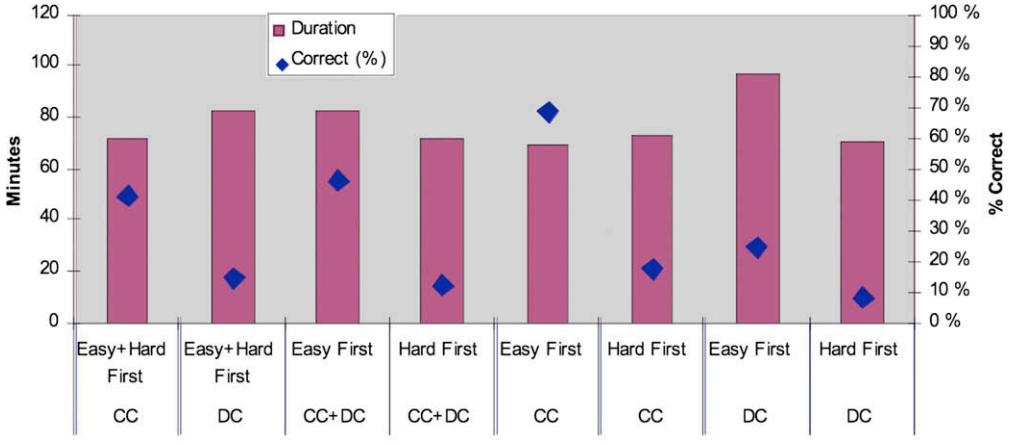

Fig. 5. The effects of design and task order on duration and correctness (task c4).

knowledge of the system will gradually become more complex. In contrast, a Hard-First task order will force the maintainer to understand most aspects of the system at once (during the first change task). However, one could argue that by using the Hard-First approach, the maintainer should have been able to create a more maintainable system, because he must have a good overview of the system in order to make the first change. For our subjects, this is not the case.

Theories of program comprehension may offer explanations of why using the CC approach and the Easy-First task order results in a more maintainable system. In Section 2 we characterised the CC-design approach and the Easy-First task order as representing a bottom-up approach to program comprehension, while the DC-design approach and the Hard-First task order represent a top-down approach to program comprehension. According to Pennington [24], when programmers are maintaining code that is completely new to them, they will first create a mental representation of a control–flow program abstraction that identifies the elementary blocks of the code and that will gradually be refined. Pennington's bottom-up model fits very well with the CC approach and Easy-First task order, where the control structure and organisation of the design are simple and the change tasks do not require a complete overview of the system. In contrast, a top-down approach to understanding a system is typically used when the code or type of code is familiar [26]. The process of understanding the program will typically consist of breaking the system down into familiar elements of that system type. Theoretically, new code could be understood using a top-down approach, but for this to be so, the maintainer must have experience of systems with similar structures. This could also be a possible explanation for why experienced programmers could benefit from the DC approach [2]. Experienced programmers have worked with many systems and can thus recognise structures (patterns) from previous work. A process of program comprehension will always try to use existing knowledge to acquire new knowledge. Existing knowledge can be classified into two types: general and software-specific [33]. The former is typically knowledge about programming languages, algorithms and such like, while the latter pertains to understanding a specific application. In our experiment, the subjects mainly had some general knowledge, but very little software-specific knowledge. Top-down program comprehension usually requires software-specific knowledge to be used successfully; hence, the lack of software-specific knowledge on the part of our subjects may explain our results. According to Von Mayrhauser and Vans, programmers with better understanding of domain are more likely to take top-down model, while those with less programming knowledge prefer bottom-up approach in program comprehension [40]. Further, Wiedenbeck and Ramalingam noticed that for small object-oriented programs, novice programmers tend to develop stronger function-related knowledge and weaker dataflow and program-related knowledge [41]. The latter describes that novice programmers focus on the low-level rather than high-level constructs.

Another possible way of explaining the result is to consider how novice programmers navigate when reading source code. Mosemann and Wiedenbeck claim that professional and novice programmers read source code in two very different ways. Professional programmers tend to read source code to understand the main structure of the system, while novices tend to read source code sequentially [42]. This claim is rooted in studies of debugging activities that found that novice programmers read source code line-by-line as they were reading a book [43,44]. The sequential method of program navigation results in a bottom-up program comprehension where the focus is details first and general structure last. As a result, novice programmers tend to gain a fragmentary knowledge that contains low-level knowledge but fails to establish the program structure, dynamic behavior, interactions and purpose [43]. The results found in this paper that shows that Easy-First outperforms Hard-First change task order in terms of correctness can be explained by the sequential navigation method most novice developers are likely to follow. When sequential program navigation is used, the Hard-First task order is less likely to succeed, as it requires more attention to the structure and interaction of the system in order to make correct changes.

Our experiment also shows that there is no significant difference in the time spent to correctly solve the tasks, regardless of task order and design style.

In summary, our results suggest that inexperienced programmers who have little prior knowledge of the system that is to be maintained are more likely to implement correct changes when using an Easy-First (as opposed to Hard-First) task order and a centralized (as opposed to a delegated) design style. The time spent on making the changes did not vary significantly by task order or design approach.

5. Threats to validity

We now discuss what we consider to be the most important threats to the validity of the experiment and how we have attempted to address the threats.

5.1. Validity of statistical conclusions

The validity of statistical conclusions concerns (1) whether the presumed cause and effect covary and (2) how strongly they covary. For the first of these inferences, one may incorrectly conclude that cause and effect covary when, in fact, they do not (a Type I error) or incorrectly conclude that they do not covary when, in fact, they do (a Type II error). For the second inference, one may overestimate or underestimate the magnitude of covariation, as well as the degree of confidence that the estimate warrants [45].

The GLM model assumptions were checked by assessing the deviance residuals [38]. For the duration models (Tables 5 and 9), plots of the deviance residuals indicated no outliers or overinfluential observations. Furthermore, a distribution analysis of the deviance residuals indicated no significant deviations from the normal distribution. Thus, the model fit was good. For the logistic models (Tables 4 and 8), a plot of the deviance residuals again indicated no potentially overinfluential observations. We also performed a Hosmer and Lemeshow Goodness-of-Fit test, which did not indicate a lack of fit ($p = 0.27$).

In summary, there are no serious threats to the validity of statistical conclusions due to lack of fit of the model. Assessments of the ANCOVA assumptions are discussed as threats to internal validity.

5.2. Internal validity

The internal validity of an experiment concerns "the validity of inferences about whether observed covariation between A (the presumed treatment) and B (the presumed outcome) reflects a causal relationship from A to B as those variables were manipulated or measured" [45]. If changes in B have causes other than the manipulation of A, there is a threat to internal validity.

The experiment was conducted in two phases and in two distinct locations, and the lack of random assignment to the TaskOrder treatment could result in skill differences between the treatment groups, which in turn would bias the results. To address this potential threat, each subject performed a pretest task, which was used to adjust for group differences by means of an analysis of covariance (ANCOVA) model [35]. An important assumption of ANCOVA is that the slope of the covariate can be considered equal

across all treatment combinations. This assumption was checked for all models. For the duration model for tasks c1–c3 (Table 5), there was in fact a significant interaction effect between pre_LogDur and TaskOrder. Thus, the ANCOVA assumption was violated in this case. As there was no statistically significant difference in the mean pretest value of the treatment groups, we could also perform a regular ANOVA, i.e., without the covariate, to check whether the violation affected the statistical conclusions.[3] The results of the ANOVA confirmed the results of the ANCOVA model reported in Table 5: no significant effects of the treatments. For the other models (Tables 4, 8 and 9), no interaction terms involving the covariate Log(pre_dur) were significant, which indicates that the homogeneity in the slopes assumption was not violated for those models. Thus, we conclude that the ANCOVA models successfully adjusted for skill differences between the treatment groups, as indicated by the pretest.

A related issue is that to perform "fair" analyses of duration, we removed subjects with incorrect solutions. The removal introduced a potential sample bias, since we removed a larger proportion of observations from the Hard-First group. Following the same arguments as above, the inclusion of the pretest in the ANCOVA models will adjust for skill differences, including the differences that were caused by removing subjects with incorrect solutions.

5.3. Construct validity

Construct validity concerns the degree to which inferences are warranted, from (a) the observed persons, settings, and cause and effect operations included in a study to (b) the constructs that these instances might represent. The question, therefore, is whether the sampling particulars of a study can be defended as measures of general constructs [45].

5.3.1. Task order

An important threat to the construct validity in our quasi-experiment is whether the ranking of the tasks are really "Easy-First" or "Hard-First", that is, whether our independent variable *TaskOrder* has adequate construct validity. In our experimental context, the difficulty level of the change tasks was determined a priori by the authors, by considering the number and complexity of classes that would be affected by each change and an estimate of the time required to perform the changes. If we consider the description of the change tasks c1, c2, and c3 (see Section 3.3.3), we can see that c1 is really a small and simple change, c2 is a bit more complicated because it involves more of the classes in the system, and c3 is a change that will affect most objects in the system. To check the construct validity, we performed a test of the time spent, correctness as well as perceived difficulty level as reported by the subjects on a 1–5 Likert scale upon completion of each of the tree tasks.

Table 12 summarizes the measures, indicating that our ranking of the tasks was adequate; both for each individual treatment group and aggregated across the treatment groups. However, it is important to note that this experiment still has only considered two specific sequences of tasks, for two small systems, which by no means is representative of all possible ways in which the independent variable "Hard-First" and "Easy-First" could have been operationalized.

5.3.2. Design

Another important threat to the construct validity concerns the extent to which the actual design alternatives used in the treatment reflect the concept studied. Since there are no operational definitions for control styles of object-oriented software, a degree of subjective interpretation is required. Another problem is to define the degree of centralization in a centralized design and the degree of decentralization in a delegated design. Ideally we should have included a whole range of systems, and quantified their degree of centralization/delegation, instead of just including two alternative designs. However, given the expert opinions in [46] and our own assessment of the designs, it is quite obvious that the DC design has a more delegated control style than has the CC design. However, it is always possible to create designs that are more centralized than CC (e.g., that consist of a single class) and a more delegated control style than the DC design. We chose to use example designs developed by others [46] as treatments that we believe are realistic and representative. By using this approach, we can avoid biased treatments and it will be easier to replicate the experiments.

5.3.3. Correctness and duration as maintainability indicators

In this experiment, two relatively simple measures were used as dependent variables. The dependent variable *correctness* was a coarse-grained, binary measure of correctness that indicates whether the subjects delivered a functionally correct solution after performing change tasks c1–c3 and c4, respectively. The dependent variable *duration* measured the time spent (excluding nonproductive time between tasks) to deliver correct solutions for change tasks c1–c3 and c4. We believe that these two variables are essential indicators of maintainability as they reflect the cost of maintenance, but there are of course many dimensions of "maintainability" that are not directly covered, e.g., the design quality of the solutions and the severity of faults. Such dimensions are however to some extent *indirectly* measured in the final benchmark task c4, since the c4 measurements depend on the overall maintainability of the solutions delivered after tasks c1–c3. For example, it is reasonable to expect that a poorly designed, but functionally correct solution for c1–c3 would have a negative impact on the time spent and correctness for the benchmark task c4, despite being functionally correct.

5.4. External validity

The issue of external validity concerns whether a causal relationship holds (1) for variations in persons, settings, treatments, and outcomes that were in the experiment and (2) for persons, settings, treatments, and outcomes that were not in the experiment [45].

5.4.1. Fatiguing effects

Despite our effort to ensure realism, the working conditions for the subjects did not represent a normal day at the office. This lack of realism was caused by the controlled environment, where, for example, the subjects were not permitted to ask others for help. In a normal working situation, it is likely that one would be less stressed and could take longer breaks than in an experimental setting. Our experiment setting/environment might cause fatigue to a degree that is not representative of realistic settings.

5.4.2. Systems and tasks

Clearly, the experimental systems in this experiment were very small compared with industrial object-oriented software systems. Furthermore, the change tasks were relatively small in size and duration. This is representative of a very common limitation in controlled software engineering experiments, but the impact of such limitations depends on the research question being asked and the extent to which the results are supported by theory that explain the underlying and more general mechanisms [12]. In this

[3] In situations where there is no difference in the mean value of the covariate (the pretest) between the treatment groups, the difference between ANCOVA and ANOVA is that the ANOVA model has a larger error term than the ANCOVA (since ANCOVA accounts for variability due to the covariate, hence reducing the error term).

Table 12
Average effort and correctness of change tasks c1–c3

Treatment group	Variable	Task c1	Task c2	Task c3
Easy-First (CC)	Correctness %	0.89	0.86	0.86
	Duration (min)	12.11	21.21	38.11
	Perceived difficulty	1.32	2.07	2.50
Easy-First (DC)	Correctness %	0.84	0.90	0.58
	Duration (min)	18.45	18.81	51.42
	Perceived difficulty	1.77	2.03	2.84
Hard-First (CC)	Correctness %	1.00	0.79	0.79
	Duration (min)	6.27	21.64	38.03
	Perceived difficulty	1.03	1.55	2.15
Hard-First (DC)	Correctness %	0.88	0.82	0.64
	Duration (min)	7.71	11.85	67.78
	Perceived difficulty	1.30	1.52	3.09
Total averages	Correctness %	0.90	0.84	0.71
	Duration (min)	10.98	18.26	49.22
	Perceived difficulty	1.35	1.78	3.22

study, we have demonstrated a clear effect of task order, despite having small systems and tasks. Generalization to larger systems and tasks can mainly be claimed based on the support of existing theories within program comprehension research. The support from several such theories is described in Section 4.3. For example, the effects of task order can be explained by the sequential navigation or bottom-up comprehension strategies most novice developers are likely to follow. When such comprehension strategies are used, the Hard-First task order is less likely to succeed, as it by its very nature requires more attention to the program flow, structure and interaction of the system in order to make correct changes. Larger programs will often have larger cognitive complexity, and may thus amplify the negative impact of sequential program navigation. Thus, we believe that the effects we have demonstrated of task order on small systems probably are *conservative* estimates of the effect that would be observed on larger systems. Still, additional experiments are required to confirm or refute this expectation.

5.4.3. Subject sample

The subject sample used in this experiment consisted of 3rd–5th year students with limited professional work experience. Based on our discussions in Section 4.3, we expect that the results would probably *not* be similar if we had used, say, a sample of senior professionals as subjects: such developers would likely be able to cope much better with the Hard-First approach, since they have sufficient experience to employ and benefit from more top-town comprehension strategies. However, as discussed in the introduction of this paper, we still believe that the scope of the study is relevant, because it is common practice to assign inexperienced developers to software maintenance tasks.

6. Conclusions

The main purpose of the experiment described in this paper was to investigate whether the order in which change tasks are sequenced can affect the maintainability of a system. Our results clearly show that inexperienced programmers with no or little prior knowledge of the system they are maintaining benefit from an *Easy-First* task order. Our results suggest that *correctness* was affected significantly by task order, while *duration* was not. However, even if our experiment did not show any direct saving of effort as a result of implementing the Easy-First task order, it is likely to save effort in the long-term. This is because it is more likely that the changes will be implemented correctly and so the system will require less maintenance. We believe this result is useful input for project managers planning software maintenance where new staff is involved or when the maintenance is being out-sourced. Our results correspond to results found in research on program comprehension, suggesting that a bottom-up (Easy-First) as opposed to a top-down (Hard-First) learning process is more appropriate for inexperienced programmers who are unfamiliar with the system to be maintained.

In this experiment, we have identified an effect of task order on maintainability with small systems and tasks. Based on existing theories on program comprehension, we have argued that the effects of task order on small systems probably are *conservative* estimates of the effect that would be observed on larger systems. Additional empirical studies are required to confirm or refute this expectation.

Furthermore, our experiment was conducted with inexperienced programmers. Theories from program comprehension research suggest that experienced programmers might actually benefit from a Hard-First over an Easy-First task order, and once again, this is an interesting area of future research. Regardless, we still believe that the scope of the study is relevant, because it is common practice to assign inexperienced developers to software maintenance tasks.

This paper presented a novel, first attempt towards obtaining a deep understanding of how the task order affects maintainability. It motivates the need for further empirical studies of how the experience of the programmers, the size of systems and tasks, and prior knowledge of a system moderates the task order effects observed in this experiment.

Acknowledgements

We thank Reidar Conradi and Magne Syrstad for help and support for carrying out the experiment. We thank Are Magnus Bruaset and Jon Bråtømyr for their excellent work on the correctness assessment of the Java solutions delivered by the subjects. We also thank Chris Wright for proofreading.

References

[1] M. Jørgensen, D.I.K. Sjøberg, Impact of experience on maintenance skills, Journal of Software Maintenance and Evolution: Research and Practice 14 (2002) 123–146.
[2] E. Arisholm, D.I.K. Sjøberg, Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software, IEEE Transactions on Software Engineering 30 (2004) 521–534.
[3] D.L. Johnson, J.G. Brodman, Applying CMM project planning practices to diverse environments, IEEE Software 17 (2000) 40–47.
[4] R.N. Charette, K.M. Adams, M.B. White, Managing risk in software maintenance, IEEE Software 14 (1997) 43–50.
[5] V. Basili, F. Shull, F. Lanubile, Building knowledge through families of experiments, IEEE Transactions on Software Engineering 24 (1999) 456–473.
[6] S. Letovsky, E. Soloway, Delocalized plans and program comprehension, IEEE Software 3 (1986) 41–49.
[7] R. Brooks, Towards a theory of the cognitive processes in computer programming, International Journal on Man–Machine Studies 9 (1977) 737–751.
[8] A. Cockburn, The coffee machine design problem: part 1 & 2, C/C++ User's Journal May/June (1998).
[9] K. Beck, W. Cunningham, A laboratory for teaching object-oriented thinking, SIGPLAN Notices 24 (1989) 1–6.
[10] R. Wirfs-Brock, B. Wilkerson, L. Wiener, Designing Object-oriented Software, Prentice Hall, New Jersey, 1990.
[11] L. Hatton, How accurately do engineers predict software maintenance tasks?, IEEE Computer 40 (2007) 64–69
[12] J.E. Hannay, D.I.K. Sjøberg, T. Dybå, A systematic review of theory use in software engineering experiments, IEEE Transactions on Software Engineering 33 (2007) 87–107.
[13] C. Ramos, K. Oliviera, N. Anquetil, Legacy software evaluation model for outsourced maintainer, in: Proceedings of Eighth Euromicro Working Conference on Software Maintenance and Reengineering, 2004, pp. 48–57.
[14] C. Poole, T. Murphy, J. Huisman, A. Higgins, Extreme maintenance, in: Proceedings of IEEE International Conference on Software Maintenance, 2001, pp. 301–309.

[15] H.M. Sneed, S. Huang, Sizing maintenance tasks for web applications. in: Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR'07), 2007, pp. 171–180.
[16] ISO9126: Information technology: software product evaluation: quality characteristics and guidelines for their use, International Organization for Standardization, 1992.
[17] E. Arisholm, Empirical Assessment of Changeability in Object-oriented Software, University of Oslo, 2001.
[18] S. Bergin, J. Keating, A case study on the adaptive maintenance of an internet application, Journal of Software Maintenance and Evolution: Research and Practice 15 (2003) 254–264.
[19] D. Grefen, S. Scheneberger, The non-homogenous maintenance periods: a case study of software modifications, in: Proceedings of IEEE International Conference on Software Maintenance, 1996, 134–141.
[20] A. Epping, C. Lott, Does software design complexity affect maintenance effort?, in: Proceedings of 19th Annual Software Engineering Workshop, 1994, 297–313.
[21] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE Transactions on Software Engineering 33 (2007) 420–432.
[22] A.J. Ko, B.A. Myers, M.J. Coblenz, H.H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Transactions on Software Engineering 32 (2006) 971–987.
[23] C. Corritore, S. Wiedenbeck, Mental representations of expert procedural and object-oriented programmers in a software maintenance task, International Journal of Human Computer Studies 50 (1999) 61–83.
[24] N. Pennington, Stimulus structures and mental representations in expert comprehension of computer programs, Cognitive Psychology 19 (1987) 295–341.
[25] S. Letovsky, Cognitive processes in program comprehension, in: Proceedings of First Workshop Empirical Studies of Programmers, 1986, pp. 58–79.
[26] E. Soloway, B. Adelson, K. Ehrlich, Knowledge and processes in the comprehension of computer programs, The Nature of Expertise (1988) 129–152.
[27] B. Schneiderman, R. Mayer, Syntactic/semantic interactions in programmer behaviour: a model and experimental results, International Journal of Computer and Information Sciences 8 (1979) 219–238.
[28] A. von Mayrhauser, A. Vans, Comprehension processes during large-scale maintenance, in: Proceedings of 16th International Conference on Software Engineering, 1994, pp. 39–48.
[29] D.C. Littman, J. Pinto, S. Letovsky, E. Soloway, Mental models and software maintenance, Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers (1986) 80–98.
[30] F. Détienne, Software design – cognitive aspects, Springer-Verlag, New York, Inc, 2002.
[31] I. Vessey, Expertise in debugging computer programs: a process analysis, International Journal of Man–Machine Studies 23 (1985).
[32] M.-A. Storey, Theories, methods and tools in program comprehension: past, present and future, in: 13th International Workshop on Program Comprehension (IWPC'05), 2005, pp. 181–191.
[33] A. von Mayrhauser, A.M. Vans, Program comprehension during software maintenance and evolution, IEEE Computer 28 (1995) 44–55.
[34] D.I.K. Sjøberg, J.E. Hannay, O. Hansen, A. Kampenes, A. Karahasanovic, N.-K. Liborg, A.C. Rekdal, A survey of controlled experiments in software engineering, IEEE Transactions on Software Engineering 31 (2005) 1–21.
[35] T.D. Cook, D.T. Campbell, Quasi-experimentation – Design & Analysis Issues for Field Settings, Houghton Mifflin Company (1979).
[36] E. Arisholm, D.I.K. Sjøberg, A controlled experiment with professionals to evaluate the effect of a delegated versus centralized control style on the maintainability of object-oriented software, Simula Technical Report, 2003.
[37] E. Arisholm, D.I.K. Sjøberg, G.J. Careluis, Y. Lindsjørn, A web-based support environment for software engineering experiments, Nordic Journal of Computing 9 (2002) 231–247.
[38] R.H. Myers, D.C. Montgomery, G.G. Vining, Generalized linear models: with applications in engineering and the sciences, Wiley-Interscience, 2001.
[39] C.F.J. Wu, M. Hamada, Experiments: Planning, Analysis, and Parameter Design Optimization, Wiley-Interscience, 2000.
[40] A. von Mayrhauser, A.M. Vans, Program understanding behavior during debugging of large scale software, in: Proceedings of 17th Workshop on Empirical Studies of Programmers, 1997, pp. 157–179.
[41] S. Wiedenbeck, V. Ramalingam, Novice comprehension of small programs written in the procedural and object-oriented styles, International Journal of Human–Computer Studies 51 (1999) 71–87.
[42] R. Mosemann, S. Wiedenbeck, Navigation and comprehension of programs by novice programmers, in: Proceedings of 9th International Workshop on Program Comprehension (WPC'01), 2001, pp. 79–88.
[43] R. Jeffries, A comparison of the debugging behavior of expert and novice programmers, A Paper Presented at the AERA Annual Meeting, 1982.
[44] M. Nanja, C.R. Cook, An analysis of the on-line debugging process, in: Empirical Studies of Programmers: Second Workshop, 1987, pp. 172–183.
[45] W.R. Shadish, T.D. Cook, D.T. Campbell, Experimental and Quasi-experimental Designs for Generalized Causal Inference, Houghton-Mifflin (2002).
[46] J.O. Coplien, A generative development-process pattern language, Pattern Languages of Program Design (1995) 183–237.