Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software

Erik Arisholm, *Member*, *IEEE*, and Dag I.K. Sjøberg, *Member*, *IEEE*

Abstract—A fundamental question in object-oriented design is how to design maintainable software. According to expert opinion, a delegated control style, typically a result of responsibility-driven design, represents object-oriented design at its best, whereas a centralized control style is reminiscent of a procedural solution, or a "bad" object-oriented design. This paper presents a controlled experiment that investigates these claims empirically. A total of 99 junior, intermediate, and senior professional consultants from several international consultancy companies were hired for one day to participate in the experiment. To compare differences between (categories of) professionals and students, 59 students also participated. The subjects used professional Java tools to perform several change tasks on two alternative Java designs that had a centralized and delegated control style, respectively. The results show that the most skilled developers, in particular, the senior consultants, require less time to maintain software with a delegated control style than with a centralized control style. However, more novice developers, in particular, the undergraduate students and junior consultants, have serious problems understanding a delegated control style, and perform far better with a centralized control style. Thus, the maintainability of object-oriented software depends, to a large extent, on the skill of the developers who are going to maintain it. These results may have serious implications for object-oriented development in an industrial context: Having senior consultants design object-oriented systems may eventually pose difficulties unless they make an effort to keep the designs simple, as the cognitive complexity of "expert" designs might be unmanageable for less skilled maintainers.

Index Terms—Design principles, responsibility delegation, control styles, object-oriented design, object-oriented programming, software maintainability, controlled experiment.

1 Introduction

A fundamental problem in software engineering is to construct software that is easy to change. Supporting change is one of the claimed benefits of object-oriented software development.

The principal mechanism used to design object-oriented software is the *class*, which enables the encapsulation of attributes and methods into logically cohesive abstractions of the world. Assigning responsibilities and collaborations among classes can be performed in many ways. In a *delegated control* style, a well-defined set of responsibilities are distributed among a number of classes [31]. The classes play specific roles and occupy well-known positions in the application architecture [32], [33]. Alternatively, in a *centralized control* style, a few large "control classes" coordinate a set of simple classes [31]. According to object-oriented design experts, a delegated control style is easier to understand and change than is a centralized control style [4], [15], [31], [32], [33].

One of the major goals of a responsibility-driven design method is to support the development of a delegated control style [31], [32], [33]; that is, the design of a delegated control style is one of its prescribed principles. The empirical study in [25] confirms that a responsibility-driven design process may result in a delegated control style. That study also suggests that a data-driven design approach (adapted from structured design to the object-oriented paradigm) results in a centralized control style because one controller class is assigned the responsibility of implementing the business logic of the application, using data from simple "data objects."

In a use-case driven design method, as advocated in most recent UML textbooks, one of the commonly prescribed principles is to assign one (central) control class to coordinate the sequence of events described by each use-case [17], [18]. However, a question not explicitly discussed in the UML textbooks is *how much* responsibility the control class should have in the design of maintainable software. At one extreme, the control class might only be responsible for *initiating* the use-case and communicating with boundary (interface) classes, while the real work is delegated to entity (business) classes, which in turn collaborate to implement the business logic and flow of events of the use-case. In this case, use-case driven design would resemble responsibility-driven design, with a delegated control style. At the other extreme, the control class might implement the actual business logic and flow of events of a use-case, in which case the entity classes function only as simple data structures with "get" and "set" methods. In this case, use-case driven design would resemble data-driven design, with a centralized control style.

• *The authors are with the Simula Research Laboratory, PO Box 134, N-1325 Lysaker, Norway. E-mail: {erika, dagsj}@simula.no.*

Manuscript received 10 Sept. 2003; revised 20 Feb. 2004; accepted 3 May 2004.
Recommended for acceptance by A. Bertolino.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0140-0903.

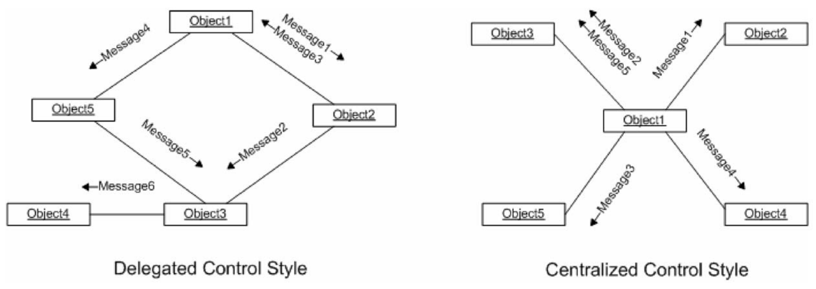

Fig. 1. Delegated versus centralized control style.

To compare the maintainability of the two control styles, the authors of this paper previously conducted a controlled experiment [2]. For the given sample of 36 undergraduate students, the delegated control style design required significantly more effort to implement the given set of changes than did the alternative centralized control style design. This difference in change effort was primarily due to the difference in the effort required to understand how to perform the change tasks.

It is evident that the expert recommendations and the results of our previous experiment run counter to each other. It might be that a delegated control style provides better software maintainability for an expert, while a centralized control style might be better for novices. Novices may struggle to understand how the objects in a delegated control style actually collaborate to fulfil the larger goals of an application. Differences in "complexity" of object-oriented designs may be explained by the cognitive models of the developers [26]. Thus, the degree of maintainability of a software application depends not only on attributes of the software artifact itself, but also on certain cognitive attributes of the particular developer whose task it is to maintain it. This factor seems to be underestimated by the object-oriented experts; neither is it investigated in most controlled experiments evaluating object-oriented technologies. Consequently, the main research question we attempt to answer in this paper is the following: For the target population of junior, intermediate, and senior software consultants with different levels of education and work experience, which of the two aforementioned control styles is easier to maintain?

We conducted an experiment with a sample of 99 Java consultants from eight consultancy companies, including the major, partly international, companies Cap Gemini Ernst & Young, Ementor, Accenture, TietoEnator, and Software Innovation. To compare differences between (categories of) professionals and students, 59 students also participated. The treatments were the same two alternative designs given in the previous pen-and-paper student experiment [2]. The experimental subjects were assigned to the two treatments using a between-subjects randomized block design.

To increase the realism of the experiment [16], [24], [29], the subjects used their usual Java development tool instead of pen and paper. The professionals were located in their usual work offices during the experiment, the students in their usual computer lab. The subjects used the Simula Experiment Support Environment [3] to receive the experimental materials, answer questionnaires, and upload task solutions. Each subject spent about one work day on the experiment. As in ordinary programming projects, the companies of the consultants were paid to participate. The students received individual payment.

The remainder of this paper is organized as follows: Section 2 outlines fundamental design principles of object-oriented software. Section 3 describes existing empirical research that evaluates object-oriented design principles. Section 4 describes the design of the controlled experiment. Section 5 presents the results. Section 6 discusses threats to validity. Section 7 concludes.

2 Delegated versus Centralized Control in Object-Oriented Designs

This section describes the concepts underlying the object of study, that is, the two control styles evaluated in the experiment. The two control styles are each illustrated by one example design, which examples are also the design alternatives used as treatments in our experiment.

2.1 Relationships between Design Properties, Principles, and Methods

To clarify the concepts studied in this paper, we distinguish between *design properties*, *design principles*, and *design methods*. Object-oriented design properties characterize the resulting design. Examples are *coupling* [8] and *cohesion* [7]. Object-oriented design principles prescribe "good" values of the design properties. Examples are *low coupling* and *high cohesion*, as advocated in [14], [22]. Object-oriented design methods prescribe a sequence of activities for creating design models of object-oriented software systems.[1] Examples are responsibility-driven design [32], data-driven design [25], [27], [30], and use-case driven design [17], [18]. Ideally, design methods should support a set of (empirically validated) design principles.

2.2 Delegated versus Centralized Control Style

The control styles studied in this paper are depicted in Fig. 1. According to the terminology defined in [31], delegated and centralized control styles embody two radically different principles for assigning responsibilities and collaborations among classes. Wirfs-Brock [31] described a delegated control style as follows:

1. The existing literature provides no clear distinction between object-oriented *analysis* and object-oriented *design*. Consequently, the process we define as object-oriented design may include activities that also might be referred to as object-oriented analysis. However, in this paper, such a distinction is not important.

TABLE 1
Overview of the Two Design Alternatives

	CC	DC
CoffeeMachine	Initiates the machine; knows how the machine is put together; handles input	Initiates the machine; knows how the machine is put together; handles input
CashBox	Knows amount of money put in; gives change; answers whether a given amount of credit is available.	Knows amount of money put in; gives change; answers whether a given amount of credit is available.
FrontPanel	Knows selection; knows price of selections, and materials needed for each; coordinates payment; knows what products are available; knows how each product is made; knows how to talk to the dispensers.	Knows selection; coordinates payment; delegates drink making to the Product.
Product		Knows its recipe and price.
ProductRegister		Knows what products are available.
Recipe		Knows the ingredients of a given product; tells dispensers to dispense ingredients in sequence.
Dispensers	Controls dispensing; tracks amount it has left.	Knows which ingredient it contains; controls dispensing; tracks amount it has left.
DispenserRegister		Knows what dispensers are available
Ingredient.		Knows its name only.
Output	Knows how to display text to the user.	Knows how to display text to the user.
Input	Knows how to receive command-line input from the user	Knows how to receive command-line input from the user
Main	Initializes the program	Initializes the program

A delegated control style ideally has clusters of well defined responsibilities distributed among a number of objects. Objects in a delegated control architecture tend to coordinate rather than dominate. Tasks may be initiated by a coordinator, but the real work is performed by others. These worker objects tend to both 'know' and 'do' things. They may even be smart enough to determine what they need to know, rather than being plugged with values via external control. To me, a delegated control architecture feels like object design at its best...

In contrast, a centralized control style typically consists of a central object (Fig. 1), which is responsible for the initiation and coordination of all tasks [31]:

A centralized control style is characterized by single points of control interacting with many simple objects. The intelligent object typically serves as the main point of control, while others it uses behave much like traditional data structures. To me, centralized control feels like a "procedural solution" cloaked in objects...

2.3 Example—The Coffee-Machine Design Problem

This section illustrates the two control styles, using two alternative example designs of the coffee-machine design problem. These designs were discussed at a workshop on object-oriented design quality at OOPSLA '97 [19] and are described in two articles in the C/C++ User's Journal [15]:

This two-article series presents a problem I use both to teach and test OO design. It is a simple but rich problem, strong on "design," minimizing language, tool, and even inheritance concerns. The problem represents a realistic work situation, where circumstances change regularly. It provides a good touch point for discussions of even fairly subtle designs in even very large systems...

The initial problem stated by Cockburn [15] was as follows:

You and I are contractors who just won a bid to design a custom coffee vending machine for the employees of Acme Fijet Works to use. Arnold, the owner of Acme Fijet Works, like the common software designer, eschews standard solutions. He wants his own, custom design. He is, however, a cheapskate. Arnold tells us he wants a simple machine. All he wants is a machine that serves coffee for 35 cents, with or without sugar and creamer. That's all. He expects us to be able to put this little machine together quickly and for little cost. We get together and decide there will be a coin slot and coin return, coin return button, and four other buttons: black, white, black with sugar, and white with sugar.

The two alternative designs discussed in [15] are, we believe, good examples of a centralized and a delegated control style, respectively. Table 1 shows the classes and their assigned responsibilities for the two alternative designs. The first design, referred to as the Centralized Control (CC) design in this paper (denoted "Mainframe design" in [15]), consists of seven classes. The second design, referred to as the Delegated Control (DC) design in this paper (denoted "Responsibility-Driven Design" in [15]), consists of 12 classes.

In both designs, the *FrontPanel* class acts as a "control class" for the use-case "Make Drink." However, the number and type of responsibilities assigned to the *FrontPanel* class are different in the two designs. In the CC design, the *FrontPanel* is responsible for most tasks: it knows the user selection, the price of each type of coffee and how each type of coffee is made. To make a specific type of coffee, the *FrontPanel* calls the *dispense* method of various *Dispenser* objects in an *if-then-else* structure. In the DC design, the

FrontPanel just *initiates* the use case, and delegates the control of how a given type of coffee is made to a *Product*, which knows its price and *Recipe*. In turn, the *Recipe* is responsible for knowing the *Ingredients* of which a product consists, but has no knowledge about pricing.

Cockburn [15] assessed the CC design as follows:

Although the trajectory of change in the mainframe approach involves only one object, people soon become terrified of touching it. Any oversight in the mainframe object (even a typo!) means potential damage to many modules, with endless testing and unpredictable bugs. Those readers who have done system maintenance or legacy system replacement will recognize that almost every large system ends up with such a module. They will affirm what sort of a nightmare it becomes.

Furthermore, Cockburn [15] assessed the DC design as follows:

The design we come up with at this point bears no resemblance to our original design. It is, I am happy to see, robust with respect to change, and it is a much more reasonable "model of the world." For the first time, we see the term "product" show up in the design, as well as "recipe" and "ingredient." The responsibilities are quite evenly distributed. Each component has a single primary purpose in life; we have avoided piling responsibilities together. The names of the components match the responsibilities.

Thus, the DC design has a distinctly delegated control style, whereas the CC design has a distinctly centralized control style. According to Cockburn [15], most novices (students) come up with the CC type of design. However, most experts would probably agree that the DC design is, as Cockburn argues, a more maintainable solution to the coffee-machine design problem.

3 Related Empirical Studies

In one of the few field experiments comparing alternative object-oriented technologies, a data-driven and a responsibility-driven design method were compared [25]. Two systems were developed based on the same requirements specification; using the data-driven and the responsibility-driven design method, respectively. The results suggest that the responsibility-driven design method results in a delegated control style, whereas the data-driven design method results in a centralized control style. Structural attribute measures (defined in [12]) of the two systems were also collected and compared. Based on the measured values, the authors suggested that use of the responsibility-driven design method resulted in higher quality software than did use of the data-driven design method because the responsibility-driven software system had less coupling and higher cohesion than did the data-driven software system. We believe it may be premature to draw such conclusions. Whether the design measures used in the experiment actually measured "quality" was not evaluated by means of direct measurement of external quality attributes.

Nevertheless, there *is* a growing body of results that indicates that class-level measures of structural attributes, such as coupling and cohesion, can be reasonably good predictors of product quality (see survey in [5]), which supports the conclusions in [25]. However, most of these metrics validation studies have been case studies and, so, there is a lack of control that limits our ability to draw conclusions regarding cause-effect relationships [20], [21]. One notable exception was a controlled experiment that investigated whether a "good" design (adhering to Coad and Yourdon's design principles [14]) was easier to maintain than was a "bad" design [6], [9]. The results suggest that reducing coupling and increasing cohesion (as suggested in Coad and Yourdon's design principles) improve the maintainability of object-oriented design documents. However, as pointed out by the authors, the results should be considered preliminary, primarily because the subjects were students with little programming experience.

A controlled experiment to assess the changeability (i.e., change effort and correctness) of the example coffee-machine designs described in Section 2.3 is reported in [2]. Thirty-seven undergraduate students were divided into two groups in which the individuals designed, coded, and tested several identical changes to one of the two design alternatives. The subjects solved the change tasks using pen and paper. Given the argumentation described in Section 2, the results were surprising in that they clearly indicated that the delegated control design requires significantly more change effort for the given set of changes than does the alternative centralized control design. This difference in change effort was primarily due to the difference in effort required to *understand* how to perform the change tasks. Consequently, designs with a delegated control style may have higher cognitive complexity than have designs using a centralized control style. No significant differences between the two designs were found with respect to correctness.

In summary, more empirical studies are needed to evaluate principles of design quality in object-oriented software development. The control style of object-oriented design represents one such fundamental design principle that needs to be studied empirically. Related empirical studies provide no convincing answers as to how the control style of object-oriented design affects maintainability. The field experiment reported in [25] lacks validation against external quality indicators. The results of the experiments in [2], [6] contain apparent contradictions. Furthermore, both experiments used students as subjects solving pen-and-paper exercises. A major criticism of such experiments is their lack of realism [16], [24], which potentially limits our ability to generalize the findings to the population about which we wish to make claims, that is, professional programmers solving real programming tasks using professional tools in a realistic development environment. An empirical study reported in [26] reveals substantial differences in how novices, intermediates, and experts perceive the difficulties of object-oriented development. These results are confirmed by a controlled experiment in which, among other things, a strong interaction between the expertise of the subjects and type of task was identified during object-oriented program comprehension [10]. Consequently, the results of the existing empirical studies are difficult to generalize to the target population of professional developers.

4 Design of Experiment

The conducted experiment was a replication of the initial pen-and-paper student experiment reported in [2]. The motivation for replicating a study is to establish an increasing range of conditions under which the findings hold, and predictable exceptions [23]. A series of replications might enable the exploratory and evolutionary creation of a theory to explain the observed effects on the object of study. In this experiment, the following three controlled factors were modified, compared with the initial experiment:

- *More representative sample of the population*—The target population of this experiment was professional Java consultants. To obtain a more representative sample of this population, we hired 99 junior, intermediate, and senior Java consultants from eight software consultancy companies. To compare differences between (categories of) professionals and students, 59 undergraduate and graduate students also participated. Descriptive statistics of the education and experience of the sample population are given in [1].
- *More realistic tools*—Professional developers use professional programming environments. Hence, traditional pen-and-paper-based exercises are hardly realistic. In this experiment, each subject used a Java development tool of their own choice, e.g., JBuilder, Forte, Visual Age, Visual J++, and Visual Café.
- *More realistic experiment environment*—The classroom environment of the previous experiment was replaced by the offices in which each developer would normally work. Thus, they had access to printers, libraries, coffee, etc., as in any other project they might be working on. The students were located in one of their usual computer labs.

4.1 Hypotheses

In this section, the hypotheses of the experiment are presented informally. The hypotheses reflect the expectation that there is an *interaction* between the programming experience and the control style of an object-oriented design. We expect experienced developers to have the necessary skills to benefit from "pure" object-oriented design principles, as reflected in a delegated control style. Based on the results of the previous experiment [2], we expect novice developers to have difficulties understanding a delegated control style and, thus, to perform better with a centralized control style. There are two levels of hypothesis: one that compares the control styles for all subjects and another that compares the relative differences between the developer categories. The null-hypotheses of the experiment are as follows:

- $H0_1$—**The Effect of Control Style on Change Effort**. The time spent on performing change tasks on the DC design and CC design is equal.
- $H0_2$—**The Effect of Control Style on Change Effort for Different Developer Categories**. The difference between the time spent on performing change tasks on the DC design and CC design is equal for the five categories of developer.
- $H0_3$—**The Effect of Control Style on Correctness**. The number of correct solutions for change tasks on the DC design and CC design is equal.
- $H0_4$—**The Effect of Control Style on Correctness for Different Developer Categories**. The difference between the number of correct solutions for change tasks on the DC design and CC design is equal for the five categories of developer.

In Section 4.5, the variables of the study are explained in more detail, and $H0_1$, $H0_2$, $H0_3$, and $H0_4$ are reformulated, the first two in terms of a GLM model, the second two in terms of a logistic regression model.

4.2 Design Alternatives Implemented in Java

The coffee-machine design alternatives described in Section 2.3 were used as treatments in the experiment. The two designs were coded using similar coding styles, naming conventions, and amount of comments. Names of identifiers (e.g., variables and methods) were long and reasonably descriptive. UML sequence diagrams of the main scenario for the two designs were given to help clarify the designs. The sequence diagrams are provided in [1].

4.3 Programming Tasks

The programming tasks of the experiment consisted of six change tasks: a training task, a pretest task, and four (incremental) coffee machine tasks ($c1 - c4$). To support the logistics of the experiment, the subjects used the Web-based Simula Experiment Support Environment (SESE) [3] to answer an experience questionnaire, download code and documents, upload task solutions, and answer task questionnaires. The experience questionnaire, detailed task descriptions, and change task questionnaire are provided in [1]. Each task consisted of the following steps:

1. Download and unpack a compressed directory containing the Java code to be modified. This step was performed only prior to task $c1$ for the coffee-machine design change tasks ($c1 - c4$) since these change tasks were based on the solution of the previous task.
2. Download task descriptions. Each task description contained a test case that each subject used to test the solution.
3. Solve the programming task using the chosen development tool.
4. Pack the modified Java code and upload it to SESE.
5. Complete a task questionnaire.

4.3.1 *Training Task*

For the training task, all the subjects were asked 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 steps outlined above.

4.3.2 *Pretest Task*

For the pretest task, all the subjects implemented the same change on the same design. The change consisted of adding

TABLE 2
Subject Assignment to Treatments Using a Randomized Block Design

	CC	DC	Total
Undergraduate	13	14	27
Graduate	15	17	32
Junior	16	15	31
Intermediate	17	15	32
Senior	17	19	36
Total	78	80	158

transaction log functionality in 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 comparing the programming skill level of the subjects. The pretest task had almost the same size and complexity as the subsequent change tasks $c1$, $c2$, and $c3$ combined.

4.3.3 Coffee-Machine Tasks

The change tasks consisted of four incremental changes to the coffee-machine:

- 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.

4.4 Group Assignment

A randomized block experimental design was used; each subject was assigned to one of two groups by means of randomization and blocking. The two groups were *CC* (in which the subjects were assigned to the CC design) and *DC* (in which the subjects were assigned to the DC design). The blocks were "undergraduate student," "graduate student," "junior consultant," "intermediate consultant," and "senior consultant." Table 2 shows the distribution of the categories of subject in the different groups.

4.5 Execution and Practical Considerations

To recruit the professional developers, several companies were contacted through their formal sales channels. A contract for payment and a time schedule were then agreed upon. The companies were paid normal consultancy fees for the time spent on the experiment by the consultants (five to eight hours each). Seniors were paid more than intermediates, who, in turn, were paid more than juniors. A project manager in each company selected the subjects from the company's pool of consultants.

To recruit the students, graduate and undergraduate students in the Department of Informatics at University of Oslo were contacted through e-mail. The students were paid a fixed amount for participating.

The experiment was conducted in 12 separate sessions on separate days (one or more sessions in each of the nine companies and one session for the students). All the subjects in a given session were colocated at the same company site. The subjects could only take breaks or make telephone calls *between* change tasks. During each session, one or several researchers were present on the site at all times, to ensure that the subjects followed our requests and to assist them in case of technical problems.

We wanted the subjects to perform the tasks with satisfactory quality in as short a time as possible, because most software engineering jobs induce a relatively high pressure on tasks to be performed. However, if the time pressure placed on the participatory subjects is too high, the quality of the task solution may be reduced to the point where it becomes meaningless to use the corresponding task times in subsequent statistical analyses. The challenge is therefore to place realistic time pressure on the subjects. The best way to deal with this challenge depends to some extent on the size, duration, and location of an experiment [28]. In this experiment, we used the following strategy:

- Instead of offering an hourly rate, we offered a "fixed" honorarium based on an estimation that the work would take five hours to complete. We told the subjects that they would be paid for those five hours independently of the time they would actually need. Hence, those subjects who finished early (e.g., in two hours) were still paid for five hours. We employed this strategy to encourage the subjects to finish as quickly as possible and to discourage them from working slowly in order to receive higher payment. However, in practice, once the five hours had passed, we told those subjects who had not finished that they would be paid for additional hours if they completed their tasks. The students received a fixed payment equivalent to eight hours salary as a teaching assistant, regardless of the actual time spent.
- The subjects were allowed to leave when they finished. Those who did not finish had to leave after eight hours.
- The subjects were informed that they were not all given the same tasks to reduce the chances that they would, for competitive reasons, prioritize speed over quality.
- The last task ($c4$) was not included in the analysis because, in our experience, the final change task in an experiment needs special attention as a result of potential "ceiling effects." If the last task is included in the analyses, it is difficult to discriminate between the performance of the subjects regarding effort and correctness. Subjects who work fast may spend more time on the last task than they would otherwise. Similarly, subjects who work slowly may have insufficient time to perform the last task correctly.

Consequently, the final change task in this experiment was not included in the analysis. Thus, the analysis of effort is not threatened by whether the subjects actually managed to complete the last task, while at the same time the presence of the large task helped to put time pressure on the subjects during the experiment. Pilot experiments were conducted to ensure that it would be very likely that all subjects would complete tasks $c1 - c3$ within a time span of a maximum of eight hours. As shown in Section 5, only two out of 158 subjects did not complete all the tasks.

We (the researchers) and the subjects signed a confidentiality agreement stating that we guaranteed that all the information about the subjects' performance should be kept strictly confidential. In particular, no information would be given to the company or to the individuals themselves about their own performance. The subjects guaranteed that they would not share information about the experiment with their colleagues, either during or after the experiment.

4.6 Analysis Model

To test the hypotheses, a regression-based approach was used on the unbalanced experiment design. The variables in the models are described below.

4.6.1 Dependent Variables

- **Log(Effort)**—the total effort in Log(minutes) to complete the change tasks. Before starting on a task, the subjects wrote down the current time. When the subjects had completed the task, they reported the total effort (in minutes) for that task. The first author of this paper double-checked the reported times using time stamps reported by the SESE tool. The variable *Effort* was the combined total effort to complete the change tasks. Thus, nonproductive time between tasks was not included. A log-transformation of the effort data gave an almost perfect normal distribution.
- **Correctness**—a binary correctness score with value 1 if all the change tasks were correctly implemented, and 0 if at least one of these tasks contained serious logical errors.

Each change task solution was reviewed by an independent consultant with a PhD in computer science who lectures on testing at the University of Oslo. He was not informed about the hypotheses of the experiment. To perform the correctness analysis, he first developed a tool that automatically unpacked and built the source code corresponding to each task solution (uploaded to SESE by the subjects). In total, this corresponds to almost 1,000 different Java programs. Then, each solution was tested using a regression test script. For each test run, the difference between the *expected* output of the test case (this test output was given to the subjects as part of the task specifications) and the *actual* output generated by each program was computed. The tool also showed the complete source code as well as the source code differences between each version of the program delivered by each subject, to identify exactly how they had changed the program to solve the change task. To perform the final grading of the task solutions, a Web-based grading tool was developed that enabled the consultant to view the source code, the source code difference, the test case output, and the test case difference. He gave the score *correct* if there were no, or only cosmetic, differences in the test case output, and no serious logical errors were revealed by manual inspection of the source code; otherwise, he gave the score *incorrect*. The consultant performed this analysis twice to avoid inconsistencies in the way he had graded the task solutions. Completing this work took approximately 200 hours.

4.6.2 Controlled Factors

- **Design**—the main treatments of the experiment, that is, the factors DC and CC.
- **Block**—the developer categories used as blocking factors in the experiment, that is, the factors Undergraduate, Graduate, Junior, Intermediate, and Senior. For the professional consultants, a project manager from each company chose consultants from the categories "junior," "intermediate," and "senior" according to how they usually would categorize (and price) their consultants. Potential threats caused by this categorization are discussed further in Section 6.1.

4.6.3 Covariates

- **Log(Pre_Dur)**—the (log-transformed) effort in minutes to complete the pretest task. The individual results of the pretest can be used as a covariate in the models to reduce the error variance caused by individual skill differences.

4.6.4 Model Specifications

For the hypotheses regarding effort, a generalized linear model (GLM) approach was used to perform a combination of analysis of variance (ANOVA), analysis of covariance (ACOVA), and regression analysis [13]. For the hypotheses regarding correctness, a logistic regression model was fitted using the same (GLM) model terms as for effort, that is, including dummy (or indicator) variables for each factor level and combinations of factor levels [13].

The models are specified in Table 3. Given that the underlying assumptions of the model are not violated, the presence of a significant model term corresponds to rejecting the related null-hypothesis. Model 1 was used to test hypotheses $H0_1$ and $H0_2$. Model 2 was used to test hypotheses $H0_3$ and $H0_4$. In addition, model 3 was included to test the hypothesis on effort restricted to those subjects with correct solutions. Thus, model 3 represents an alternative way to assess the effect of the design alternatives on change effort. Since the subjects with correct solutions no longer represent a random sample, the covariate *Log(Pre_Dur)* was included to adjust for skill differences between the groups. Furthermore, since the covariate is confounded with *Block*, it is no longer meaningful to include *Block* in model 3.

The final specification of the models must take place after the actual analyses because the validity of the

TABLE 3
Model Specifications

Model	Response	Model Term	Primary use of model term
(1)	Log(Effort)	**Design**	**Test H0$_1$ (Effort Main Effect)**
		Block	Assess the effect of different developer categories on effort
		Design* Block	**Test H0$_2$ (Effort Interaction)**
(2)	Correct	**Design**	**Test H0$_3$ (Correctness Main Effect)**
		Block	Assess the effect of different developer categories on correctness
		Design* Block	**Test H0$_4$ (Correctness Interaction)**
(3)	Log(Effort)	**Design**	**Alternative Test of H0$_1$ for subjects with correct solutions**
		Log(Pre_Effort)	Covariate to adjust for programming skill differences
		Log(Pre_Effort)*Design	Test on homogeneity of slopes

TABLE 4
Descriptive Statistics of Change Effort (in Minutes) and Correctness (in Percent)

Block	Design	N	N*	Mean	Std	Min	Q1	Median	Q3	Max	Correct
Undergraduate	CC	13	0	79	30	45	56	81	87	161	62%
	DC	14	0	108	63	23	73	88	151	267	29%
		27	0	94	51	23	60	84	99	267	44%
Graduate	CC	15	0	65	23	23	49	60	85	105	80%
	DC	17	0	73	37	23	52	63	85	173	65%
		32	0	69	31	23	51	63	85	173	72%
Junior	CC	16	0	95	32	39	76	95	114	170	63%
	DC	15	0	110	46	60	71	102	127	217	33%
		31	0	102	39	39	72	100	122	217	48%
Intermediate	CC	17	0	107	49	51	72	91	133	215	65%
	DC	14	1	101	46	54	63	92	127	202	40%
		31	1	104	47	51	69	91	126	215	53%
Senior	CC	16	1	103	62	35	64	75	135	253	76%
	DC	19	0	71	38	31	40	61	95	169	74%
		35	1	86	52	31	51	67	111	253	75%
Total	CC	77	1	91	44	23	60	83	101	253	69%
	DC	79	1	91	48	23	60	77	120	267	50%
		156	2	91	46	23	60	82	105	267	59%

underlying model assumptions must be checked against the actual data. For example, we determined that a log-transformation of effort was necessary to obtain models with normally distributed residuals, which is an important assumption of GLM. Furthermore, the inclusion of insignificant interaction terms may affect the validity of the coefficients and p-values (and the resulting interpretation) of other model terms. Insignificant interaction terms are therefore candidates for removal from the model. Whether insignificant terms should actually be removed depends on whether the reduced model fits the data better than the complete model. This is explained further in Section 5.

5 Results

This section describes the results of the experiment. In Section 5.1, descriptive statistics of the data are provided to illustrate the size and direction of the effects of the experimental conditions. In Section 5.2, the hypotheses outlined in Section 4.1 are tested formally using the statistical models described in Section 4.6. Finally, in Section 5.3, we draw some general conclusions by interpreting both the descriptive statistics and the results from the formal hypothesis tests.

5.1 Descriptive Statistics

Table 4 shows the descriptive statistics related to the main hypotheses of the experiment. Two of the 158 subjects in the experiment did not complete all the tasks, as indicated by column N^*. The columns *Mean* to *Max* show the descriptive statistics of the change effort (in minutes to solve change tasks c1 + c2 + c3). The column *Correct* shows the percentage of the subjects that delivered correct solutions for all three tasks. The *Total* row shows that the mean time required to perform the tasks was 91 minutes for both the CC and DC design. Furthermore, 69 percent of the subjects delivered correct solutions on the CC design, but only 50 percent did on the DC design.

However, there are quite large differences between the different *categories* of developer, especially when comparing undergraduate and junior developers with graduate students and senior professionals. The apparent interaction between developer category and design alternative is illustrated in Fig. 2. For example, the undergraduate students spent on average about 30 percent less time on

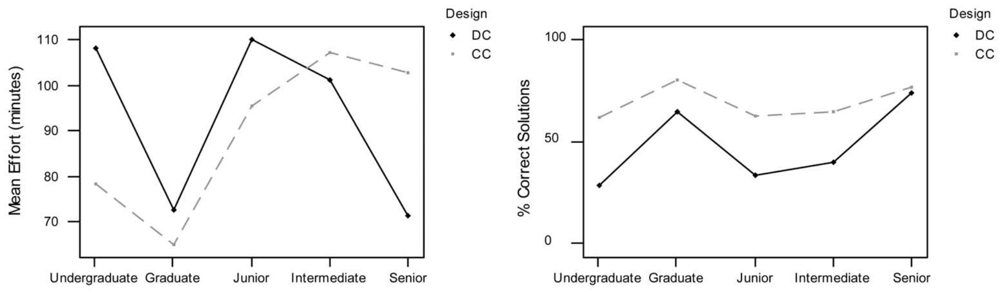

Fig. 2. Interaction plots of mean effort and correctness.

TABLE 5
GLM Model (Model 1) for Log(Effort) (Hypotheses $H0_1$ and $H0_2$)

```
Factor  Type Levels Values
Design  fixed     2 DC CC
Block   fixed     5 Undergraduate Graduate Junior Intermediate Senior

Analysis of Variance for Log(Effort), using Adjusted SS for Tests

Source         DF     Seq SS     Adj SS     Adj MS       F      P
Design          1     0.0310     0.0004     0.0004    0.00  0.964
Block           4     4.0454     3.9932     0.9983    4.84  0.001
Design*Block    4     1.4788     1.4788     0.3697    1.79  0.133
Error         146    30.1090    30.1090     0.2062
Total         155    35.6642

Term                    Coef   SE Coef        T      P
Constant             4.39952   0.03657   120.31  0.000
Design
    DC               0.00166   0.03657     0.05  0.964
Block
    Graduate        -0.25945   0.07224    -3.59  0.000
    Junior           0.16559   0.07303     2.27  0.025
    Intermediate     0.15855   0.07326     2.16  0.032
    Senior          -0.08006   0.06999    -1.14  0.255
Design*Block
    DC*Graduate      0.3012   0.07224     0.42  0.677
    DC*Junior        0.05869   0.07303     0.80  0.423
    DC*Intermediate -0.02752   0.07326    -0.38  0.708
    DC*Senior       -0.17024   0.06999    -2.43  0.016
```

the CC design than on the DC design (79 minutes versus 108 minutes). They were also much more likely to produce correct solutions on the CC design than on the DC design (62 percent versus 29 percent). This indicates that, for undergraduate students, the CC design is easier to change than is the DC design. This picture is reversed when considering the seniors: they spent on average about 30 percent *more* time on the CC design than on the DC design (103 minutes versus 71 minutes). For the seniors, there is no difference in correctness for the two design alternatives (76 percent for the CC design versus 74 percent for the DC design). This indicates that, for senior developers, the DC design is easier to change than is the CC design. The graduate students, juniors, and intermediates seem to benefit from using the CC design when considering both change effort and correctness, although the differences between the control styles are much smaller than for the undergraduate students and seniors.

5.2 Hypothesis Tests

The results of testing the hypotheses on change effort are shown in Table 5. Residual analyses of the model indicate that the assumptions of the GLM model are not violated (further details are provided in [1]).

There is insufficient evidence to reject the null-hypothesis $H0_1$, that is, we cannot conclude that there is a difference in change effort between the two design alternatives (*Design*, p = 0.964). By contrast, the results identify significant differences in change effort for the five developer categories (*Block*, p = 0.001). A posthoc Tukey's pairwise comparison of differences in the mean of the developer categories show that, overall, the graduate students were faster than juniors (p = 0.0027) and intermediates (p = 0.0034). There were no significant differences between any other pair of categories.

Regarding the hypotheses about the interaction between design and developer category, $H0_2$, there is insufficient support for rejecting the null-hypothesis (*Design*Block*, p = 0.133). In light of the descriptive statistics, this was somewhat surprising because the relative difference in change effort between the design alternatives is still considerable for *some* categories of developer and in opposite directions. Since we did not have a clear hypothesis about how much experience would be required to

TABLE 6
Logistic Regression Model (Model 2) for Correctness (Hypotheses $H0_3$ and $H0_4$)

```
Response Information

Variable  Value       Count
Correct   1              94  (Event)
          0              64
          Total         158

Logistic Regression Table
                                                          Odds        95% CI
Predictor             Coef    SE Coef         Z      P   Ratio    Lower    Upper
Constant            0.2403     0.4330      0.55 0.579
Design
 DC                -0.9154     0.3483     -2.63 0.009     0.40     0.20     0.79
Block
 Graduate           1.2307     0.5667      2.17 0.030     3.42     1.13    10.39
 Junior             0.1342     0.5422      0.25 0.805     1.14     0.40     3.31
 Intermediate       0.3196     0.5386      0.59 0.553     1.38     0.48     3.96
 Senior             1.3941     0.5606      2.49 0.013     4.03     1.34    12.10

Log-Likelihood = -97.814
Test that all slopes are zero: G = 17.675, DF = 5, P-Value = 0.003

Goodness-of-Fit Tests

Method                Chi-Square    DF       P
Pearson                    1.526     4  0.822
Deviance                   1.486     4  0.829
Hosmer-Lemeshow            1.500     6  0.959
```

benefit from the DC design, the analysis model including all developer categories in the interaction term is quite conservative. Only large interaction effects between several of the categories would result in a significant model term. Consequently, we performed a posthoc analysis based on model 1, but included only two developer categories: subjects with "high experience," which included the seniors and intermediates, and subjects with "low experience," which included the undergraduates, graduates, and juniors. In this case, the interaction term was significant (p = 0.028). We emphasize that this is an exploratory analysis because the experience level was not set a priori, but was instead set on the basis of the actual data.

The results of testing the hypotheses on correctness are shown in Table 6. The results clearly show that the subjects are much less likely to produce correct solutions on the DC design than on the CC design (*Design*, odds-ratio = 0.40, p = 0.009), all other conditions being equal. The null-hypothesis $H0_3$ is rejected. Furthermore, graduate students and seniors are much more likely to produce correct solutions (odds-ratios 3.42 and 4.03, respectively) than are the other developer categories. The interaction term *Design*Block* was removed from the logistic regression model because the coefficients were far from significant and reduced the goodness of fit. Hence, there is insufficient statistical evidence to reject $H0_4$. We cannot conclude that the CC design improves correctness for only *some* categories of developers. On the basis of the evidence collected, it improves correctness for all the categories. The goodness-of-fit tests for the model in Table 6 show a high correlation between the observations and the model estimates. Thus, the underlying model assumptions of logistic regression are not violated.

Finally, Table 7 shows the results of the analysis of covariance model on Log(Effort) for the subjects who managed to produce correct solutions. The results show that the change effort is much less for the DC design than for the CC design (*Design*, p = 0.003). Thus, those subjects who actually manage to understand the DC design sufficiently well to produce correct solutions also use less time than those who produce correct solutions on the CC design. As can be seen from the descriptive statistics (Table 4) and from the logistic regression model of correctness (Table 6), these subjects are overrepresented by senior consultants and graduate students. Residual analyses of the model indicate that the assumptions of the GLM model are not violated. See details in [1].

5.3 Summary of Results

This section summarizes the results by considering both the descriptive statistics (which describe the results of the specific sample population) and the hypotheses tests (which indicate the extent to which the results can be generalized to the target population) with regard to both effort and correctness.

Based on the formal hypothesis tests, the results suggest that there is no difference in change effort between the two designs when considering all subjects, regardless of whether they produced correct solutions or not (model 1). The descriptive statistics indicate large relative differences between two specific categories of developer (undergraduate student versus senior consultant), but there is insufficient support for an interaction effect between the design alternatives and the given developer categories with regard to effort (p = 0.133). However, a posthoc analysis conducted on the basis of the actual data still suggests that there is an interaction effect between a more coarse-grained variable "experience" and change effort (p = 0.028).

All developer categories are more likely to produce correct solutions on the CC design than on the DC design

TABLE 7
Change Effort for Subjects with Correct Solutions

```
Factor     Type Levels Values
Design    fixed      2 DC CC

Analysis of Variance for Log(Effort), using Adjusted SS for Tests

Source           DF     Seq SS     Adj SS     Adj MS       F      P
Log(pre_Effort)  1      3.2835     3.1802     3.1802   24.06  0.000
Design           1      1.2421     1.2421     1.2421    9.40  0.003
Error           91    12.0275    12.0275     0.1322
Total           93    16.5531

Term             Coef      SE Coef        T      P
Constant         2.9912    0.2622     11.41  0.000
Log(pre_Effort)  0.32893   0.06706     4.91  0.000
Design
  DC            -0.11628   0.03793    -3.07  0.003
```

(model 2). There is no support for an interaction effect between design alternatives and the developer category with regard to correctness. However, the effect size of design on correctness is very large for the undergraduate students and junior developers, who clearly have serious difficulty in producing correct solutions on the DC design, whereas the effect size of design is negligible for the seniors.

When only considering those subjects who managed to produce correct solutions (probably the most skilled subjects because the subjects with correct solutions also used, on average, considerably *less* time than did subjects with incorrect solutions), the DC design seems to require less effort than does the CC design (model 3).

In summary, when considering both effort and correctness in combination, the results suggest the following conclusions. Only senior consultants seem to have the necessary skills to benefit from the DC design. The graduate students also perform well on the DC design, but they perform even better on the CC design. The CC design favors the less skilled developers, overrepresented by undergraduate students and junior developers. There are no clear indications in either direction for the intermediate developers.

6 Threats to Validity

This paper reports an experiment with a high degree of realism compared with previously reported controlled experiments within software engineering. Our goal was to obtain results that could be generalized to the target population of professional Java consultants solving real programming tasks with professional development tools in a realistic work setting. This is an ambitious goal, however. For example, there is a trade off between ensuring realism (to reduce threats to *external* validity) and ensuring control (to reduce threats to *internal* validity). This section discusses what we consider to be the most important threats to the validity of this experiment.

6.1 Construct Validity

The construct validity concerns whether the independent and dependent variables accurately measure the concepts we intend to study.

6.1.1 Classification of the Control Styles

An important threat to the construct validity in this experiment is the extent to which the actual design alternatives that were used as treatments ("delegated" versus "centralized" control styles) are representative of the concept studied. There is no operational definition to classify precisely the control style of object-oriented software; a certain degree of subjective interpretation is required. Furthermore, when considering the extremes, the abstract concepts of a centralized and delegated control style might not even be representative of realistic software designs. Still, some software systems might be "more centralized than" or "more delegated than" others.

Based on expert opinions in [15] and our own assessment of the designs, it is quite obvious that the DC design has a more delegated control style than the CC design. However, it is certainly possible to design a coffee-machine with an even more centralized control style than the CC design (e.g., a design consisting of only one control class and no entity classes whatsoever), or a more delegated control style than the DC design. We chose to use, as treatments, example designs developed by others [15]. We believe these treatments constitute a reasonable trade off between being clear representatives of the two control styles, and being realistic and unbiased software design alternatives.

6.1.2 Classification of Developers

Someone who is considered as (say) an intermediate consultant in one company might be considered (say) a senior in another company. Thus, the categories are not necessarily representative of the categories used in every consultancy company. A replication in other companies might therefore produce different results with respect to how the variable *Block* affects change effort and correctness. However, as seen from the results, the *Block* factor representing the categories is a significant explanatory variable of change effort and correctness, and, as expected, senior consultants provided better solutions in a shorter time than did juniors and undergraduate students. Thus, for the purpose of discriminating between the programming skill and experience of the developers, the classification was sufficiently accurate.

6.1.3 Measuring Change Effort

The effort measure was affected by noise and disturbances. Some subjects (in particular the professionals) might have been more disturbed or have taken longer breaks than did others. For example, senior consultants are likely to receive more telephone calls because they typically have a central role in the projects in which they would normally participate. To address this possible threat, we instructed the consultants not to answer telephone calls or talk to colleagues during the experiment. The subjects were also instructed to take their lunch break only *between* two change tasks. At least one of the authors of this paper was present at the company site during all experiment sessions and observed that these requests were followed to a large extent. The monitoring functionality of SESE [3] also enabled us to monitor the progress of each subject at all times, and follow up if we observed little activity. Similar measures were applied during the student experiment session.

6.1.4 Measuring Correctness

The dependent variable *Correct* was binary, and indicated whether the subjects produced functionally correct solutions on *all* the change tasks, thus producing a working final program. As described in Section 4.6, a significant amount of effort was spent on ensuring that the correctness scores were valid. More complex measures to identify the *number* of programming faults or the *severity* of programming faults were also considered. However, such measures would necessarily be more subjective and, hence, more difficult to use in future replications than the adopted "correct"/"not correct" score.

6.1.5 Effort and Correctness as Indicators of Maintainability

An important issue is whether one of the designs, after being subject to the changes, would be more "maintainable," or, in general, have higher "quality" than the other design, for certain categories of developer. We believe that the effort spent when performing the changes, and the achieved correctness, represent two important indicators of the maintainability of the two control styles. However, due to the limited number and duration of tasks, they only indicate short-term maintainability. For example, the results reported in [2] suggest that the change tasks on the CC design result in higher coupling and require more lines of code to be changed than do the change tasks on the DC design. As argued in [2], these internal attributes indicate that the DC design might be more structurally stable in the long run, but we will still not really know what the consequences would be regarding the actual costs of maintaining the software, unless new experiments are performed. Thus, this is a threat to construct validity that also has consequences for the external validity of this experiment.

6.2 Internal Validity

The internal validity of an experiment is the degree to which conclusions can be drawn about the causal effect of the controlled factors on the experimental outcome.

6.2.1 Differences in Settings between Developer Categories

To improve external validity, the setting of the experiment should be as realistic as possible [28], [29]. Thus, the students in this experiment were situated in a computer lab; the professional consultants in a normal work environment. Furthermore, each developer was permitted to use a Java development environment of their own choice. Most of the students used Emacs and Javac, whereas the professionals used a variety of professional integrated development environments. Finally, there were differences in payment between students and professionals. We cannot rule out the possibility that these differences in settings are confounding factors regarding a direct comparison of the performance of students versus professionals.

However, the primary goal of this experiment was to compare *relative* differences of the effect the two control styles, for which all categories of student and professional developer were evenly distributed across the two design alternatives. Furthermore, there were no differences in setting *within* each developer category. Hence, these differences in setting were not included as additional covariates in the models described in Section 4.6. For example, although the students and professionals used different tools, the distribution of tools was quite even across the two design alternatives and within each developer category. In this particular case, we also checked the extent to which the chosen development tool affected the performance of the subjects, by including *DevelopmentTool* as a covariate in the models described in Section 4.6. The term was not a significant explanatory variable for effort ($p = 0.437$) or correctness ($p = 0.347$). Another possibility would have been to use only one specific tool to completely eliminate variations due to different tools as a possible confounding factor. However, that would introduce other threats, related, for example, to tool learning effects.

In summary, we believe it is unlikely that the main results described in this paper; that is, the relative comparison of the effect of the two design alternatives for different categories of developer are threatened by differences in setting.

6.3 External Validity

The question of external validity concerns "[the] populations, settings, treatment variables, and measurement variables [to which] this effect [can] be generalized" [11].

6.3.1 Scope of Systems and Tasks

Clearly, the two alternative designs in this experiment were very small compared with "typical" object-oriented software systems. Furthermore, the change tasks were also relatively small in size and duration. However, the change task questionnaires received from the participants after they had completed the change tasks indicate that the *complexity* of the tasks was quite high. Nevertheless, we cannot rule out the possibility that the observed effects would be different if the systems and tasks had been larger.

The scope of this study is limited to situations in which the maintainers have no prior knowledge of a system. It is possible that the results do not apply to situations in which the maintainers are also the original designers. As also discussed in Section 6.1, a related issue is whether the short-term effects observed in this experiment are representative of long-term maintenance. It is possible that the effects we observed are due principally to the higher cognitive complexity of a delegated control style, and that even less skilled maintainers will eventually pass the learning curve of a delegated control style to the extent that they can benefit from it.

6.3.2 Fatiguing Effects

Despite our effort to ensure realism, the experiment is still not completely representative of a "normal day at the office." In a normal work situation, one might be able to take longer breaks and in general be less stressed and tired than in an experimental setting. We cannot rule out the possibility that such fatiguing effects might introduce a bias for one of the control styles.

6.3.3 Representativeness of Sample

An important question for this experiment is whether the professional subjects were representative of "professional Java consultants." Our sample included consultants from major international software consultancy companies. A project manager was hired from each company to, among other things, select a representative sample of their consultants for the categories "junior," "intermediate," and "senior." The selection process corresponded to how the companies would usually categorize and price consultants. Hence, in addition to experience and competence, availability was also one of the selection criteria. Thus, it could be the case that the "best" professionals were underrepresented in our sample since it is possible that they had already been hired by other companies. Fortunately, we observed that, on many occasions, the project managers took busy consultants off their current projects to participate in the experiment.

7 Conclusions

The degree of maintainability of a software application depends not only on attributes of the software itself, but also on certain cognitive attributes of the particular developer whose task it is to maintain it. This aspect seems to be underestimated by expert designers. Most experienced software designers would probably agree that a delegated control style is more "elegant" and a better object-oriented representation of the problem to be solved, than is a centralized control style. However, care should be taken to ensure that future maintainers of the software are able to understand this (apparently) elegant design. If the cognitive complexity of a design is beyond the skills of future maintainers, they will spend more time, and probably introduce more faults, than they would with a (for them) simpler but less "elegant" object-oriented design.

Assuming that it is not only highly skilled experts who are going to maintain an object-oriented system, a viable conclusion from the controlled experiment reported in this paper is that a design with a centralized control style may be more maintainable than is a design with a delegated control style. These results are also relevant with regard to a use-case driven design method, which may support both control styles: It is mainly a question of how much responsibility is assigned to the control class of each use case.

Although an important goal of this experiment was to ensure realism, by using a large sample of professional developers as subjects who are instructed to solve programming tasks with professional development tools in a normal office environment, there are several threats to the validity of the results that should be addressed in future replications. Increasing the realism (and, thereby, external validity) reduced the amount of control, which introduced threats to internal validity. For example, we allowed the developers to use a development tool of their own choice. Another possibility would have been to use only one specific tool to eliminate variations due to different tools as a possible confounding factor. However, that would introduce other threats, related, for example, to tool learning effects. Thus, we believe that this reduction in control is a small price to pay considering that the improved realism of this experiment allows us to generalize the results beyond that which would be possible in a more controlled laboratory setting with students solving pen-and-paper tasks. Still, whether the results of this experiment generalize to realistically sized systems and tasks is an open question. Consequently, the most important means to improve the external validity of the experiment is to increase the size of the systems and the tasks. Furthermore, the results might be different in situations where less skilled maintainers successfully perform a sufficient number of tasks to pass the learning curve of a delegated control style.

The results of this experiment are surprising in a further way. For the given tasks, the graduate students performed very well, and outperformed junior and intermediate consultants. One reason could be that the project managers selected low-skilled consultants for the experiment. However, contrary to this, the project managers confirmed that they also included their best people, and on many occasions took them off their current projects to participate in the experiment. A more likely reason is that graduate students because of the stringent selection process for admission to Masters programs, are better than relatively inexperienced professionals. Another possibility is that formal training in object-oriented programming is more important than work experience. Both hypotheses are, to some extent, supported by the descriptive statistics presented in [1]: The seniors are more likely to have completed graduate studies than are the juniors and intermediates. Both the seniors and graduate students have more credits in computer science courses than have the undergraduate students, juniors, and intermediate consultants. In future studies, we will attempt to explore these complex interactions among the underlying characteristics (such as programming experience in specific languages, work experience, and education) to better explain the observed variations in programmer performance.

ACKNOWLEDGMENTS

The authors thank Lionel Briand, Magne Jørgensen, Vigdis By Kampenes, Ray Welland, Chris Wright, and the anonymous reviewers for their valuable contributions to this paper. They thank KompetanseWeb for their excellent support on the SESE tool. Gunnar Carelius provided valuable support during the preparation and quality assurance of the experimental materials in SESE. Are Magnus Bruaset did an outstanding job on the testing and qualitative assessment of the Java solutions delivered by the subjects. They thank the students at the University of Oslo for their participation, and the staff in the Department of Informatics at the same university for their technical support. Finally, this paper would not have been possible without the consultants and project managers who participated from the following companies: Accenture, Cap Gemini Ernst & Young, Ementa, Ementor, Genera, Object-net, Software Innovation, Software Innovation Technology, and TietoEnator.

REFERENCES

[1] E. Arisholm and 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," Technical Report 2003-6, Simula Research Laboratory, http://www.simula.no/erika, 2003.

[2] E. Arisholm, D.I.K. Sjøberg, and M. Jørgensen, "Assessing the Changeability of Two Object-Oriented Design Alternatives—A Controlled Experiment," *Empirical Software Eng.*, vol. 6, no. 3, pp. 231-277, 2001.

[3] E. Arisholm, D.I.K. Sjøberg, G.J. Carelius, and Y. Lindsjørn, "A Web-Based Support Environment for Software Engineering Experiments," *Nordic J. Computing*, vol. 9, no. 4, pp. 231-247, 2002.

[4] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *SIGPLAN Notices*, vol. 24, no. 10, pp. 1-6, 1989.

[5] L.C. Briand and J. Wust, "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers*, vol. 59, pp. 97-166, 2002.

[6] L.C. Briand, C. Bunse, and J.W. Daly, "A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs," *IEEE Trans. Software Eng.*, vol. 27, no. 6, pp. 513-530, 2001.

[7] L.C. Briand, J.W. Daly, and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.*, vol. 3, no. 1, pp. 65-117, 1998.

[8] L.C. Briand, J.W. Daly, and J. Wust, "A Unified FrameWork for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.*, vol. 25, no. 1, pp. 91-121, 1999.

[9] L.C. Briand, C. Bunse, J.W. Daly, and C. Differding, "An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents," *Empirical Software Eng.*, vol. 2, no. 3, pp. 291-312, 1997.

[10] J.-M. Burkhardt, F. Detienne, and S. Wiedenbeck, "Object-Oriented Program Comprehension: Effect of Expertice, Task and Phase," *Empirical Software Eng.*, vol. 7, no. 2, pp. 115-156, 2002.

[11] D.T. Campell and J.C. Stanley, *Experimental and Quasi-Experimental Designs for Research.* Rand McNally and Company, 1963.

[12] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, 1994.

[13] R. Christensen, *Analysis of Variance, Design and Regression.* Chapman & Hall/CRC Press, 1998.

[14] P. Coad and E. Yourdon, *Object-Oriented Design*, first ed. Prentice-Hall, 1991.

[15] A. Cockburn, "The Coffee Machine Design Problem: Part 1 & 2," *C/C++ User's J.*, May/June 1998

[16] R.L. Glass, "The Software Research Crisis," *IEEE Software*, vol. 11, no. 6, pp. 42-47, 1994.

[17] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process.* Addison-Wesley, 1999.

[18] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[19] R.K. Keller, A. Cockburn, and R. Schauer, "Object-Oriented Design Quality: Report on OOPSLA '97 Workshop #12," *Proc. OOPSLA '97 Workshop Object-Oriented Design Quality,* http://www.iro.umontreal.ca/keller/Workshops/OOPSLA97, 1997.

[20] B. Kitchenham, L. Pickard, and S.L. Pfleeger, "Case Studies for Method and Tool Evaluation," *IEEE Software*, vol. 12, no. 4, pp. 52-62, 1995.

[21] B.A. Kitchenham, "Evaluating Software Engineering Methods and Tools. Part 1: The Evaluation Context and Evaluation Methods," *ACM Software Eng. Notes*, vol. 21, no. 1, pp. 11-15, 1996.

[22] K.J. Lieberherr and I.M. Holland, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, vol. 6, no. 5, pp. 38-48, 1989.

[23] R.M. Lindsay and A.S.C. Ehrenberg, "The Design of Replicated Studies," *The Am. Statistician*, vol. 47, no. 3, pp. 217-228, 1993.

[24] C. Potts, "Software Engineering Research Revisited," *IEEE Software*, vol. 10, no. 5, pp. 19-28, 1993.

[25] R.C. Sharble and S.S. Cohen, "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods," *Software Eng. Notes*, vol. 18, no. 2, pp. 60-73, 1993.

[26] S.D. Sheetz, "Identifying the Difficulties of Object-Oriented Development," *J. Systems and Software*, vol. 64, no. 1, pp. 23-36, 2002.

[27] S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press, 1988.

[28] D.I.K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, and M. Vokác, "Challenges and Recommendations when Increasing the Realism of Controlled Software Engineering Experiments," *Empirical Methods and Studies in Software Eng. (ESERNET 2001-2002)*, R. Conradi and A.I. Wang, eds., 2003.

[29] D.I.K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. Koren, and M. Vokác, "Conducting Realistic Experiments in Software Engineering," *Proc. First Int'l Symp. Empirical Software Eng. (ISESE '2002)*, pp. 17-26, Oct. 2002.

[30] S. Tockey, B. Hoza, and S. Cohen, "Object-Oriented Analysis: Building on the Structured Techniques," *Proc. Software Improvement Conf.*, 1990.

[31] R.J. Wirfs-Brock, "Characterizing your Application's Control Style," *Report on Object Analysis and Design*, vol. 1, no. 3, 1994.

[32] R.J. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility Driven Approach," *SIGPLAN Notices*, vol. 24, no. 10, pp. 71-75, 1989.

[33] R.J. Wirfs-Brock, B. Wilkerson, and R. Wiener, *Designing Object-Oriented Software.* Prentice-Hall, 1990.

Erik Arisholm received the MSc degree in electrical engineering from the University of Toronto and the PhD degree in computer science from the University of Oslo. He has seven years industry experience in Canada and Norway as a lead engineer and design manager. He is now a researcher in the Department of Software Engineering at the Simula Research Laboratory and an associate professor in the Department of Informatics at the University of Oslo. His main research interests are object-oriented design principles and methods, static and dynamic metrics for object-oriented systems, and methods and tools for conducting controlled experiments in software engineering. He is a member of the IEEE and IEEE Computer Society.

Dag I.K. Sjøberg received the MSc degree in computer science from the University of Oslo in 1987 and PhD degree in computing science from the University of Glasgow in 1993. He has five years industry experience as consultant and group leader. He is now the research manager of the Department of Software Engineering at the Simula Research Laboratory and a professor of software engineering in the Department of Informatics at the University of Oslo. Among his research interests are research methods in empirical software engineering, software process improvement, software effort estimation and object-oriented analysis, and design. He is a member of the IEEE and IEEE Computer Society.