Simula Research Laboratory Technical Report 2003-6

A Controlled Experiment with Professionals to Evaluate the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software

Erik Arisholm and Dag I.K. Sjøberg

{erika,dagsj}@simula.no

Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker
NORWAY

Abstract. One 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 take part 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 having 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. The results may have serious implications for object-oriented development in an industrial context: having senior consultants design object-oriented systems that eventually will be maintained by juniors may be unwise, since the cognitive complexity of such "expert" designs might be unmanageable for less skilled maintainers.

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*, enabling 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 [29]. The classes play specific roles and occupy well-known positions in the application architecture [30, 31]. Alternatively, in a *centralized control* style, a few, large "control classes" co-ordinate a set of simple classes [29]. According to object-oriented design experts, a delegated control style is easier to understand and change than is a centralized control style [3, 13, 29-31].

One of the major goals of a responsibility-driven design method is to support the development of a delegated control style [29-31], that is, the design of a delegated control style is one of its prescribed principles. The empirical study in [23] 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 [15, 16]. However, a question not explicitly discussed in the UML textbooks is *how much* responsibility the control class should have to design 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.

To compare the maintainability of the two control styles, the authors of this paper previously conducted a controlled experiment [1]. 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 effort required to understand how to perform the change tasks.

Consequently, there is a contradiction between the expert recommendations and the results of our previous experiment. 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 [24]. Thus, software maintainability is not only an attribute of the software artefact; it is also an attribute of the actual developer changing the software. 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 [1]. The experimental subjects were assigned to the two treatments using a between-subjects randomized block design.

To increase the realism of the experiment [14, 22, 27], 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 [2] 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 remainder of this paper is organized as follows. Section 2 outlines fundamental design principles of object-oriented software. Section 3 describes existing empirical research evaluating 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. Two example designs illustrate the two control styles, respectively. These 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* [7] and *cohesion* [6]. Object-oriented design principles prescribe "good" values of the design properties. Examples are *low coupling* and *high cohesion,* as advocated in [12, 20]. Object-oriented design methods prescribe a sequence of activities for creating design models of object-oriented software systems.[1] Examples are responsibility-driven design [30], data-driven design [23, 25, 28] and use-case driven design [15, 16]. Ideally, design methods should support a set of (empirically validated) design principles.

[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.

2.2. Delegated versus Centralized Control Style

The control styles studied in this paper are depicted in Figure 1. According to the terminology defined in [29], delegated and centralized control styles embody two radically different principles for assigning responsibilities and collaborations among classes. A delegated control style is described as follows:

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…

Wirfs-Brock [29]

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

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…

Wirfs-Brock [29]

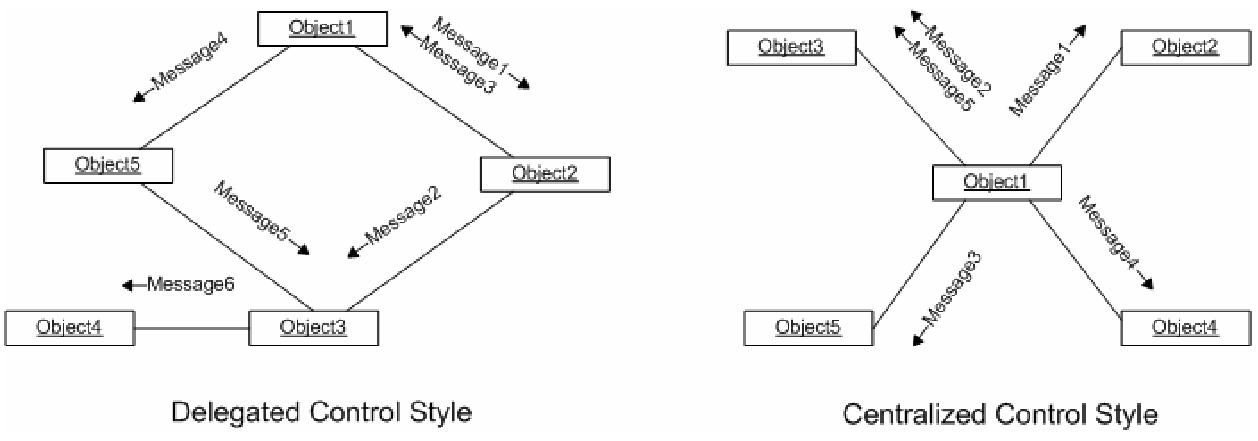


Figure 1. Delegated versus Centralized Control Style

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 [17] and are described in two articles in C/C++ User's Journal [13]:

> *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…*
>
> Cockburn [13]

The initial problem statement 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.*
>
> Cockburn [13]

The two alternative designs discussed in [13] 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 [13]), consists of seven classes. The second design, referred to as the Delegated Control (DC) design in this paper (denoted "Responsibility-Driven Design" in [13]), consists of twelve classes.

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

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 [13] 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 [13] 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 [13], 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 [23]. 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 [10]) 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 produced 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 empirically validated by direct measurement of external quality attributes.

Nevertheless, there *is* a growing body of results indicating that class-level measures of structural attributes such as coupling and cohesion can be reasonably good predictors of product quality (see survey in [4]), hence supporting the conclusions in [23]. Most of these metrics validation studies have been case studies. Thus, there is a lack of control that limits our ability to draw conclusions regarding cause-effect relationships [18, 19]. One notable exception was a controlled experiment that investigated whether a "good" design (adhering to Coad and Yourdon's design principles [12]) was easier to maintain than was a "bad" design [5, 8]. 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 [1]. 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. With regards to correctness, no significant differences between the two designs were found.

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 [23] lacks validation against external quality indicators. The results of the experiments in [1, 5] 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 [14, 22], 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 [24] 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, amongst others, a strong interaction between the expertise of the subjects and type of task were identified during object-oriented program comprehension [9]. 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 [1]. The motivation for replicating a study is to establish an increasing range of conditions under which the findings hold, and predictable exceptions [21]. 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 sample population are given in Appendix A.

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

This section informally describes the hypotheses of the experiment. They 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 [1], we expect novice developers to have difficulties understanding a delegated control style, and to thus perform better on a centralized control style. There are two levels of hypothesis: One comparing the control styles for all subjects; the other comparing the relative differences between the developer categories. The null-hypotheses of the experiment are:

$H0_1$ – 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. Furthermore, $H0_1$ and $H0_2$ are reformulated in terms of a GLM model and $H0_3$ and $H0_4$ in terms of a logistic regression model.

4.2. Design Alternatives Implemented in Java

The coffee-machine design alternatives explained 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 (Appendix E).

4.3. Programming Tasks

The programming tasks of the experiment consisted of six change tasks: a training task, a pre-test 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) [2] to answer an experience questionnaire, download code and documents, upload task solutions and answer task questionnaires. 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 task solution of the previous task.
2. Download task descriptions (Appendix F). Each task description contained a test case that each subject used to test the solution.
3. Solve the programming task using their chosen development tool.
4. Pack the modified Java code and upload it to SESE.
5. Complete a task questionnaire (Appendix G).

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 a reverse order. The purpose of this task was to familiarize the subjects with the steps outlined above.

Pre-test Task

For the pre-test task, all the subjects implemented the same change on the same design: it consisted of adding 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 pre-test task had almost the same size and complexity as the subsequent change tasks *c1, c2* and *c3* combined.

Coffee-Machine Tasks

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

c1. *Implement a coin return-button*.

c2. *Make bouillon as a new type of drink.*

c3. *Check whether all ingredients are available for the selected drink.*

c4. *Make your own drink by selecting among the available ingredients*.

Special Last Task

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 (*c4*) was not included in the analysis.

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". The descriptive statistics of the subjects are given in Appendix A. Table 2 shows the distribution of the categories of subject in the different groups.

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

4.5. Execution and Practical Considerations

The companies were paid normal consultancy fees for the time spent on the experiment by the consultants (five to eight hours each). A project manager in each company selected the subjects from the company's pool of consultants.

We wanted the subjects to perform the tasks with satisfactory quality in as short a time as possible, as most software engineering jobs put a relatively high pressure on tasks to be performed. However, if the time pressure put on the participatory subjects is too high, then 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 put 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 [26]. 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 subjects were allowed to leave when they finished.
- 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 was not included in the analysis.

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.

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). 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. A log-transformation of the effort data gave a nearly 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 1000 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.

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.

Covariates

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

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 [11]. 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 [11].

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 underlying model assumptions has to be checked on the basis of 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 p-values (and interpretation) of other model terms and should therefore be considered removed. Whether insignificant model terms should be removed depends on whether the reduced model fits the data better than the complete model. This is explained further in Section 5.

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

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 overall 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 the 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 is 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 Figure 2. For example, the undergraduate students spent on average about 30 percent less time on 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.

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%

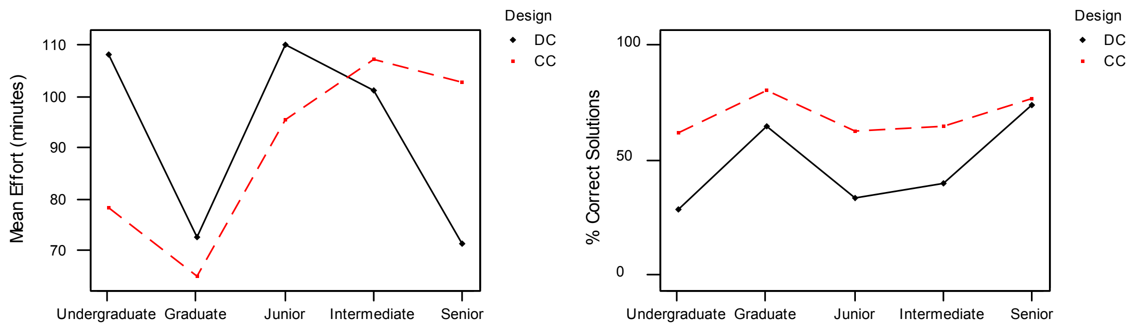


Figure 2. Interaction plots of mean effort and correctness

5.2. Hypothesis Tests

The results of testing the hypotheses on change effort are shown in Table 5. 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 =

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       0.03012   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
```

0.001). Regarding the hypotheses on the interaction between design and developer category, $H0_2$, there is weak support for rejecting the null-hypothesis (*Design*Block*, p = 0.133). However, looking at the individual coefficients for the interaction term, we may conclude that seniors are not faster than undergraduate students when considering the combined results of both the DC and CC designs (*Senior*, p = 0.255), but that the seniors spend significantly less time on the DC design than on the CC design compared with undergraduate students (*DC*Senior*, p = 0.016). The size of this interaction effect can be seen from the graphic representation of the descriptive statistics in Figure 2. Appendix C shows the residual analysis of the model, indicating that the assumptions of the GLM model are not violated.

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-

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
```

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; 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. Thus, those subjects who actually *manage*

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
```

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. Appendix D shows the residual analysis of the model, indicating that the assumptions of the GLM model are not violated.

5.3. Summary of Results

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. However, there is an interaction between the design alternatives and the developer categories with regards to effort, particularly when comparing senior consultants with undergraduate students. Furthermore, the interaction *effect size* is considerable, as illustrated by the descriptive statistics: undergraduate students (and juniors) use on average 30 percent *less* time on the CC design, whereas seniors use on average 30 percent *more* time on the CC design.

All developer categories are more likely to produce correct solutions on the CC design than on the DC design. There is no support for an interaction effect between design alternatives and the developer category with regards 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 the subjects who managed to produce correct solutions (probably the most skilled subjects because the subjects with correct solutions also on average used considerably *less* time than did subjects with incorrect solutions), the DC design seems to require less effort than does the CC design. However, since those subjects are over-represented by the seniors, this model confirms the following overall conclusion: the DC design favors the most highly skilled developers, over-represented by senior developers, whereas the CC design favors the less skilled developers, over-represented by undergraduate students and junior developers. There are no clear indications in either direction when considering both effort and correctness for the intermediate developers or the graduate students.

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.

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 representatives 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 [13] 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 [13]. 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.

Classification of Developers

It is likely that someone who would be considered as (say) an intermediate consultant in one company would 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 regards 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 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.

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 phone calls because they typically have a central role in the projects they would normally participate in. To address this possible threat, we instructed the consultants not to answer phone calls or talk to colleagues during the experiment. The subjects were also instructed to take the 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 thus observed that these requests were followed to a large extent. The monitoring functionality of SESE [2] also enabled us to monitor the progress of each subject at all times, and follow up if we observed little activity.

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 discriminating 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.

6.2. 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.

Instrumentation Differences between Developer Categories

The students in this experiment were situated in a computer lab, but the professional consultants were situated in a normal work environment while participating in the experiment. We cannot rule out that this difference in setting between the students and professionals introduced a threat with regards to the validity of the comparison between students and professionals. For example, one might argue that the professionals would feel less time pressure than would the students. Based on our observations, we believe this is not the case; both students and professionals apparently worked very hard.

Development Tools

To increase the realism (and external validity), we decided that each developer could 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. As a result of the randomized block design, the distribution of tools was quite even across the two design alternatives. Furthermore, we 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 4.5. The term was not a significant explanatory variable for effort ($p = 0.437$) or correctness ($p = 0.347$). Thus, it is unlikely that the chosen tools introduced a bias for one of the designs.

6.3. External Validity

The external validity of the experiment concerns whether the results can be generalized to a realistic development context [26, 27].

Size and Complexity of 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 questionnaires received from the participants after they had completed the change tasks (Appendix G) indicate that the *complexity* of the tasks was quite high. Still, we cannot rule out that the effects we observed when comparing the control styles would be different if the systems and tasks had been larger.

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 others things, select 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 there is a likelihood that they had already been hired by other companies. To address this threat, our agreement with the companies stated that the project manager should select a representative sample from their consultants. Fortunately, we observed that the project managers were quite eager to also include "busy" Java consultants.

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 regards 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, thereby adding a confounding factor. However, 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 what 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 still 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.

Acknowledgements

We thank Lionel Briand, Magne Jørgensen, Vigdis By Kampenes, Ray Welland and Chris Wright for their valuable contributions to this paper. We 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. We thank the students at Univ. of Oslo for their participation, and the staff at Dept. of Informatics 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, Genera, Cap Gemini Ernst & Young, Ementa, Ementor, Software Innovation, Software Innovation Technology (Sweden), Objectnet and TietoEnator.

References

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

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

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

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

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

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

[7] L. C. Briand, J. W. Daly, and J. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91-121, 1999.

[8] 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 Engineering*, vol. 2, no. 3, pp. 291-312, 1997.

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

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

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

[12] P. Coad and E. Yourdon, *Object-Oriented Design*, First ed: Prentice-Hall, 1991.

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

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

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

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

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

[18] 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.

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

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

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

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

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

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

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

[26] 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," In Reidar Conradi and Alf Inge Wang (Eds.): *"Empirical Methods and Studies in Software Engineering – Experiences from the ESERNET project", Forthcoming as a Springer Verlag LNCS,* July 2003.

[27] 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. ISESE'2002 (First International Symposium on Empirical Software Engineering), October 3-4, 2002*, pp. 17-26, 2002.

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

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

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

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

Appendix A. Descriptive Statistics of the Subjects

Variable	Block	N	Mean	Median	StDev	Min	Max	Q1	Q3
Age (years)	Undergrad	27	25.074	24	4.009	22	38	23	26
	Graduate	32	25.813	25	3.167	23	37	24	27
	Junior	31	27.548	27	5.403	22	54	25	28
	Intermed	32	30.61	28	8.28	22	62	26	31
	Senior	36	32	30	6.05	24	54	28.25	34.5
Work Exp (years)	Undergrad	27	2.593	1	4.06	0	15	0	4
	Graduate	32	2.625	2	3.452	0	18	0	3.75
	Junior	31	2.871	1	4.808	0	27	1	4
	Intermed	32	5.75	3	7.73	0	35	2	5
	Senior	36	7.611	6.5	5.463	0	27	4	10
Programming Exp (years)	Undergrad	27	1.074	0	2.129	0	10	0	2
	Graduate	32	1.219	0	3.19	0	18	0	1
	Junior	31	1.533	1	4.1	0	23	0	1
	Intermed	32	4.5	2	6.67	0	26	1	4
	Senior	36	6.278	5	5.38	0	27	3.25	8.75
Education (years)	Undergrad	27	3.154	3.15	1.11	1.25	5.5	2.3	3.65
	Graduate	32	4.378	3.95	1.035	3.25	7.85	3.8	4.975
	Junior	31	4.065	4	1.184	0.25	6.25	3.4	5
	Intermed	32	4.153	4.1	1.687	0	10	3.063	5
	Senior	36	4.011	4	2.422	0	14	3	5
CS Education (years)	Undergrad	27	1.2556	1.25	0.3881	0.5	2.2	1	1.5
	Graduate	32	1.616	1.5	0.593	0.55	3	1.25	2
	Junior	31	1.334	1	0.958	0.05	4	0.5	2
	Intermed	32	1.478	1.375	1.008	0	3.5	0.6	2
	Senior	36	1.749	1.5	1.145	0	4	1	2.5
Java (LOC)	Undergrad	27	20400	10000	26942	10	100000	5000	20000
	Graduate	32	54484	8000	177812	500	1000000	3000	20000
	Junior	31	4478	2000	9029	0	50000	500	5000
	Intermed	32	6819	4000	10374	1	55000	1000	10000
	Senior	36	28497	5000	83964	0	500000	625	23750
C++ (LOC)	Undergrad	27	1553	25	4286	0	20000	0	500
	Graduate	32	9415	1000	35272	0	200000	50	4750
	Junior	31	1935	500	2962	0	10000	100	2000
	Intermed	32	1169	550	2079	0	10000	0	1000
	Senior	36	36299	1000	166132	0	1000000	425	9000
Total LOC	Undergrad	27	43185	19500	50914	3275	200001	11000	67500
	Graduate	32	129093	19925	401407	6500	2260000	10888	65625
	Junior	31	48643	12500	127894	300	556300	4000	22850
	Intermed	32	40360	19400	45517	5	160500	10400	53550
	Senior	36	141850	45500	415313	0	2410000	11500	78875
UML Exp (1-5)	Undergrad	27	3.074	3	0.781	1	4	3	4
	Graduate	32	2.5	2.5	0.95	1	4	2	3
	Junior	31	2.516	3	0.926	1	4	2	3
	Intermed	32	2.563	2.5	0.982	1	5	2	3
	Senior	36	2.944	3	0.893	1	5	2	4

Appendix B. Experience Questionnaire

(Translated from Norwegian. The actual questionnaire was implemented in the SESE tool)

Date of birth:
Java development environment you will use in this experiment:

<u>WORK EXPERIENCE</u>
Years programming work experience:
Years total work experience:

<u>EDUCATION</u>
Number of credits in computer science courses:
Number of total university credits:

<u>PROGRAMMING SKILL AND EXPERIENCE</u>
Please rate your general programming skills (1: Novice – 5: Expert):

Please rate your Java programming skills (1: Novice – 5: Expert):
Approximately how many lines of Java code you have written:

Please rate your C++ programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of C++ code you have written:

Please rate your Simula programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of C++ code you have written:

Please rate your SmallTalk programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of SmallTalk code you have written:

Please rate your C programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of C code you have written:

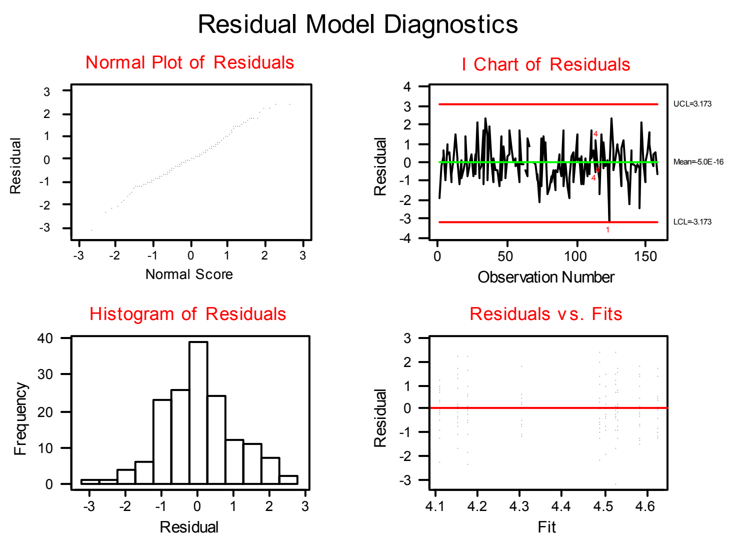
Please rate your Pascal programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of Pascal code you have written:

Please rate your [   ] programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of code you have written in this language:

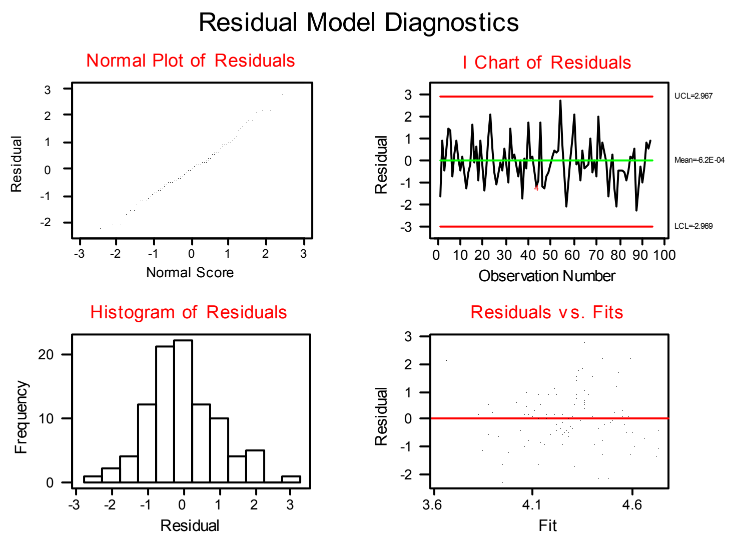
Please rate your [   ] programming skills (1: Novice – 5: Expert):
Give an estimate of how many lines of code you have written in this language:

<u>DESIGN METHOD KNOWLEDGE:</u>
UML/Rose (1: Novice – 5: Expert):
OMT (1: Novice – 5: Expert):
Responsibility-Driven Design (1: Novice – 5: Expert):
CRC (1: Novice – 5: Expert):
Role modelling (1: Novice – 5: Expert):
Structured Analysis and/or Structured Design (1: Novice – 5: Expert):
Data Driven/Relational Database Design (1: Novice – 5: Expert):
Other method [   ] (1: Novice – 5: Expert):
Other method [   ] (1: Novice – 5: Expert):

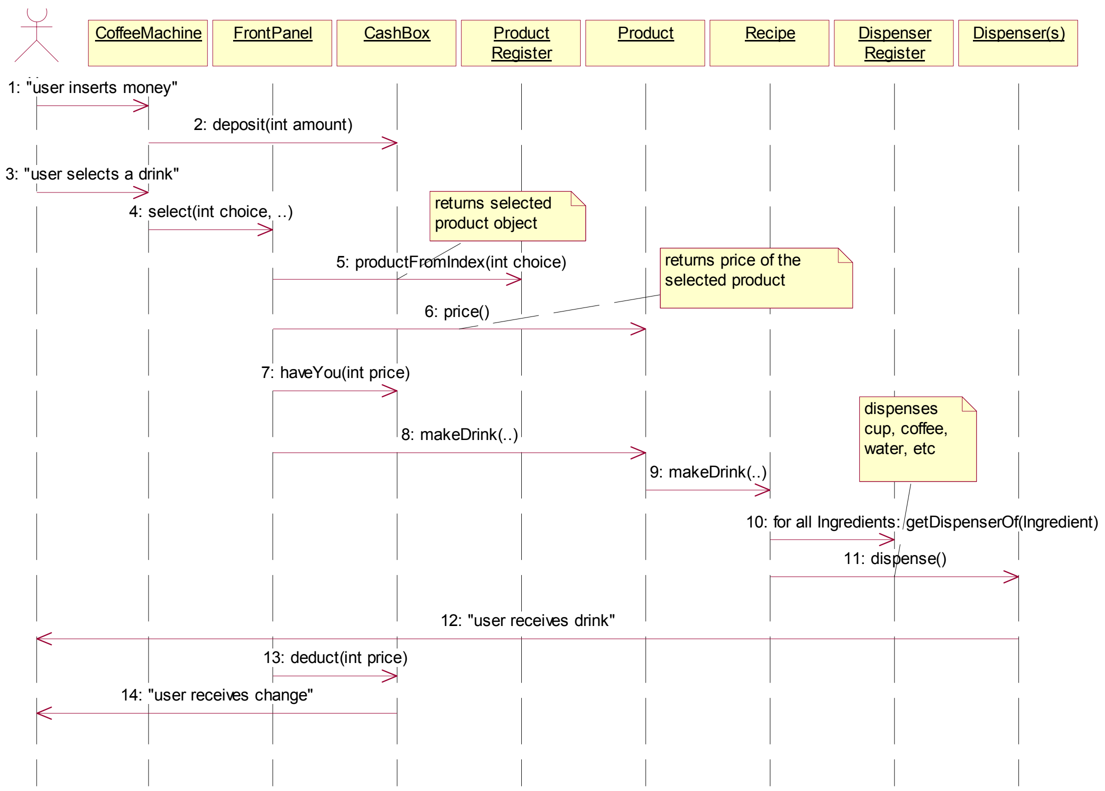
Appendix C. Residual Analysis of Model (1)



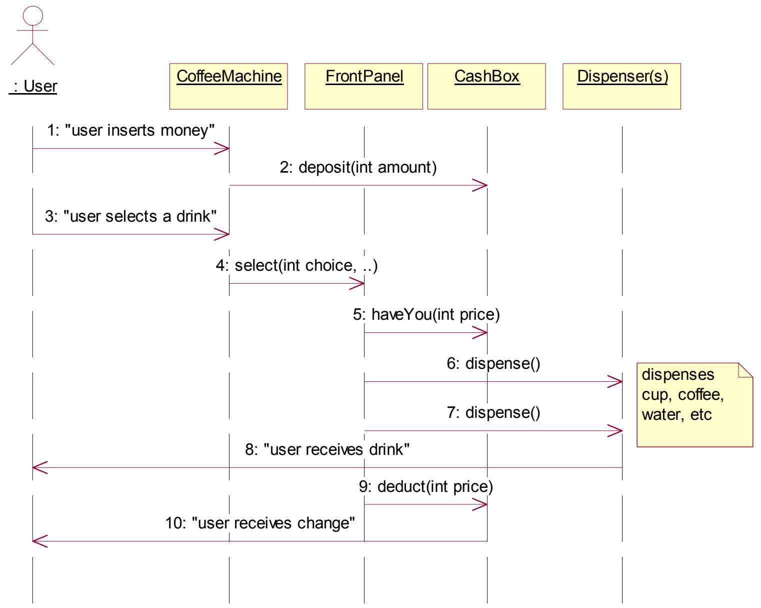
Appendix D. Residual Analysis of Model (3)



Appendix E. Sequence Diagrams of the Design Alternatives


CoffeeMachine
FrontPanel
CashBox
Product Register
Product
Recipe
Dispenser Register
Dispenser(s)
1: "user inserts money"
2: deposit(int amount)
3: "user selects a drink"
4: select(int choice, ..)
returns selected product object
5: productFromIndex(int choice)
returns price of the selected product
6: price()
7: haveYou(int price)
8: makeDrink(..)
dispenses cup, coffee, water, etc
9: makeDrink(..)
10: for all Ingredients: getDispenserOf(Ingredient)
11: dispense()
12: "user receives drink"
13: deduct(int price)
14: "user receives change"

Sequence Diagram for the DC Design



Sequence Diagram for the CC Design

Appendix F. Change Task Descriptions
(Some details are omitted. Translated from Norwegian)

TRAINING TASK

Complete the code you just downloaded so that it can read an arbitrary number of lines of text from INPUT and stores each string in a Vector. When the user presses <CR>, the program should write the number of lines of text, and thereafter print out the text in the reverse order (that is, the last string should be printet first).

Test case:

```
Enter a string. Finish with <CR>
abc
Enter next string. Finish with <CR>
def
Enter next string. Finish with <CR>
ghi
Enter next string. Finish with <CR>

You entered 3 strings.
The strings in reverse order are:
ghi
def
abc
```

PRE-TEST TASK

The code you just downloaded contains a simple automated teller machine (ATM). At present, the ATM has the following functionality:

- **New account:** … (detailed description omitted in this report)
- **Withdraw:** …(detailed description omitted in this report)
- **Deposit**: … (detailed description omitted in this report)

Add the following functionality to the ATM:

- **Account Statement:** Gives an account statement for a customer (menu choice = "Statement"). For every withdrawal a given customer has made, the statement should contain a line "Withdrew <amount>. Similarly, for every deposit the statement should contain a line "Deposited <amount>. Then the current account balance is printed. For details, refer to the following test case.

Test case:

```
Menu: N = New account W=Withdraw. D=Deposit. S=Statement Q=Quit
N
Enter a new account number:
per hansen
Please enter a personal pin code:
1234
New account has been created.
Menu: N = New account W=Withdraw. D=Deposit. S=Statement Q=Quit
D
Enter your account number:
per hansen
Enter your pin code:
1234
Insert money:
200
Menu: N = New account W=Withdraw. D=Deposit. S=Statement Q=Quit
W
Enter your account number:
per hansen
Enter your pin code:
1234
Enter amount:
100
Dispensing 100
Menu: N = New account W=Withdraw. D=Deposit. S=Statement Q=Quit
W
Enter your account number:
per hansen
Enter your pin code:
1234
Enter amount:
50
Dispensing 50
Menu: N = New account W=Withdraw. D=Deposit. S=Statement Q=Quit
S
Enter your account number:
per hansen
Enter your pin code:
1234
Deposited 200
Withdrew 100
Withdrew 50
-------------------------------
Account balance 50
Menu: N = New account W=Withdraw. D=Deposit. S=Statement Q=Quit
```

CHANGE TASK c1

In this task, you shall extend the coffee machine with a "return button" functionality that returns the deposited funds. The menu choice is called "Return".

Test Case:

```
Menu: I=insert S=select R=return Q=quit
I
Amount>
4
    CashBox: Depositing 4
    You now have 4 credits.

Menu: I=insert S=select R=return Q=quit
R
    CashBox: Returning 4

Menu: I=insert S=select R=return Q=quit
```

CHANGE TASK c2

In this task, you shall extend the machine to make bouillon. Bouillon costs more than coffee. While coffee costs 5 credits, bouillon costs 6 credits.

Test Case:

```
Menu: I=insert S=select R=Return Q=quit
I
Amount>
6
    CashBox: Depositing 6
    You now have 6 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream, 5=Bouillon)>
5
    Dispensing cup
    Dispensing bouillon
    Dispensing water
    CashBox: Returning 0

Menu: I=insert S=select R=Return Q=quit
```

CHANGE TASK c3

Unfortunately, there is a quite serious problem with the coffee machine at present. If the user chooses for example "coffee with cream", and the cream dispenser is empty, the machine gives a small error message, after which it dispenses black coffee (without cream). If the machine does not contain any more cups, the machine dispenses the drink right into the drain… The user will of course get quite irritated over having to pay for this!

The simplest solution to this problem is that the user receives a message if the machine is out of a required ingredient of the selected drink. Then, the user is given the option to choose another drink. The following test case illustrates what should happen when the machine runs out of cream:

Test Case:

```
Menu: I=insert S=select R=Return Q=quit
I
Amount>
5
    CashBox: Depositing 5
    You now have 5 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream, 5=Bouillon)>
2
    Dispensing cup
    Dispensing coffee
    Dispensing water
    Dispensing cream <after this the machine is out of cream>
    CashBox: Returning 0

Menu: I=insert S=select R=Return Q=quit
I
Amount>
5
    CashBox: Depositing 5
    You now have 5 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream, 5=Bouillon)>
2
        Sorry, no more cream! Select another.

Menu: I=insert S=select R=Return Q=quit
```

CHANGE TASK c4

You are going to make a new menu choice “Make your own drink”, which allows the customer to choose among any combination of available ingredients to make a custom drink (see test-case). Note! There is no checking on whether the combination of ingredients “makes sense”. However, if the machine is (or becomes) empty of a given ingredient, the customer should receive an error message and can then choose an alternative ingredient. Each shot of an ingredient costs 2 credits. If the customer has put on insufficient amounts of money for the chosen set of ingredients, the customer receives the message "Insufficient funds" and thereafter the menu choice “Menu: I=insert S=select R=Return Q=quit”.

Test Case:

```
Menu: I=insert S=select R=Return Q=quit
I
Amount
10
    CashBox: Depositing 10
    You now have 10 credits.

Menu: I=insert S=select R=Return Q=quit
S
Select Drink (1 = Black Coffee, 2=Coffee w/Cream, 3=Coffee w/Sugar, 4=Coffee w/Sugar & Cream 5= Bouillon, 6=Make your
own drink)>
6
Select Ingredient (1=Cup, 2=Coffee, 3=Sugar, 4=Water, 5=Cream, 6=Bouillon, 0=Make Drink)
1
    You have selected cup
    This drink costs 2 credits
Select Ingredient (1=Cup, 2=Coffee, 3=Sugar, 4=Water, 5=Cream, 6=Bouillon, 0= Make Drink)
2
    You have selected cup, coffee
    This drink costs 4 credits
Select Ingredient (1=Cup, 2=Coffee, 3=Sugar, 4=Water, 5=Cream, 6=Bouillon, 0= Make Drink)
2
    You have selected cup, coffee, coffee
    This drink costs 6 credits
Select Ingredient (1=Cup, 2=Coffee, 3=Sugar, 4=Water, 5=Cream, 6=Bouillon, 0= Make Drink)
4
    You have selected cup, coffee, coffee, water
    This drink costs 8 credits
Select Ingredient (1=Cup, 2=Coffee, 3=Sugar, 4=Water, 5=Cream, 6=Bouillon, 0= Make Drink)
5
    Sorry, no more cream!
    You have selected cup, coffee, coffee, water
    This drink costs 8 credits
Select Ingredient (1=Cup, 2=Coffee, 3=Sugar, 4=Water, 5=Cream, 6=Bouillon, 0= Make Drink)
0
    Dispensing cup
    Dispensing coffee
    Dispensing coffee
    Dispensing water
    CashBox: Returning 2

Menu: I=insert S=select R=Return Q=quit
```

Appendix G. Change Task Questionnaire

Time (hh:mm) when starting of the change task:

Time (hh:mm) when completing the change task:

Effort (in minutes) to solve the change task:

A. Effort to understand how to solve the change task:
B. Effort to code the change task:
C. Effort to evaluate/test the solution (run test-case):

How would you characterize your strategy to solve the task?
Very explorative (1) – Very systematic (5):

What is your subjective assessment of the quality of your solution?
Very poor (1) - Very good (5):

How confident are you that the solution does not contain serious faults?
Very unsure (1) - Very confident (5):

How difficult did you think the change task was?
Very easy (1) - Very difficult (5):

Other comments: