Empirical Software Engineering, 9, 149–195, 2004.

A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment

MAREK VOKÁČ marekv@simula.no
Simula Research Laboratory, N-1325, Lysaker, Norway

WALTER TICHY tichy@ira.uka.de
Universität Karlsruhe, Postfach 6980, D-76128 Karlsruhe, Germany

DAG I. K. SJØBERG dagsj@simula.no
Simula Research Laboratory, N-1325, Lysaker, Norway

ERIK ARISHOLM erika@simula.no
Simula Research Laboratory, N-1325, Lysaker, Norway

MAGNE ALDRIN magne.aldrin@nr.no
Norwegian Computing Center, P.O. Box 114 Blindern, N-0314 Oslo, Norway

Editor: Dieter Rombach

Abstract. Software "design patterns" seek to package proven solutions to design problems in a form that makes it possible to find, adapt and reuse them. To support the industrial use of design patterns, this research investigates when, and how, using patterns is beneficial, and whether some patterns are more difficult to use than others. This paper describes a replication of an earlier controlled experiment on design patterns in maintenance, with major extensions. Experimental realism was increased by using a real programming environment instead of pen and paper, and paid professionals from multiple major consultancy companies as subjects.

Measurements of elapsed time and correctness were analyzed using regression models and an estimation method that took into account the correlations present in the raw data. Together with on-line logging of the subjects' work, this made possible a better qualitative understanding of the results.

The results indicate quite strongly that some patterns are much easier to understand and use than others. In particular, the Visitor pattern caused much confusion. Conversely, the patterns Observer and, to a certain extent, Decorator were grasped and used intuitively, even by subjects with little or no knowledge of patterns.

The implication is that design patterns are not universally good or bad, but must be used in a way that matches the problem and the people. When approaching a program with documented design patterns, even basic training can improve both the speed and quality of maintenance activities.

Keywords: Controlled experiment, design patterns, real programming environment, qualitative results.

1. Introduction

Design patterns have become quite popular (Buschmann et al., 1996; Gamma et al., 1995). In addition to making design knowledge available to both junior and more experienced developers, it is claimed that design patterns define a common terminology that can be used to document the design. According to the classic books by Alexander (1978, 1987), individual patterns can be combined into a language that guides the designer. This should simplify communication of the underlying design and assumptions from the original designers to maintainers of the software.

An expected benefit—because design patterns tend to provide solutions that are more complete than just solving the immediate problem at hand—is the ability to add functionality at a later time without causing major changes. The same property may, however, introduce unneeded complexity.

Prechelt et al. (2001) performed a controlled experiment in late 1997 to measure the effects of using several design patterns in a maintenance situation. They used 29 unpaid professionals from a single company as subjects, and used pen and paper for the programming exercises. Based on the properties of the four selected patterns, they hypothesized both positive and negative effects from the patterns.

Their results generally agreed with expectations. The use of the observer pattern in a simple program had the expected negative effect on maintainability; the Visitor pattern was neutral in a context where a negative effect was expected. The Decorator pattern had the expected positive effect, and AbstFactory caused only small differences.

The authors of the present paper replicated their experiment with 44 paid, professional subjects using the same programs in a real programming environment, instead of pen and paper. This increases the experimental realism and, thereby, the applicability of the results.

The technical environment also allowed us to collect more data, making more detailed analysis and inferences possible. It also allowed us to address some of the threats to validity of the original experiment, such as effects of subjects' C++ knowledge, and of actually programming and testing the solutions. At the same time, performing a close replication allowed us to do a direct comparison of our results to those of the original experiment.

Our results reinforce the conclusion that each design pattern has its own nature and proper place of use; they cannot be classified as "good" or "bad" in general terms. We found a positive effect for Observer and a very strong negative effect for Visitor, while Decorator and AbstFactory found effects similar to those of the original experiment.

The remainder of this paper is organized as follows: Section 2 describes the original experiment. The present replication is described in Section 3. Section 4 contains the programs, work tasks, hypotheses and a summary of the quantitative results. Section 5 discusses the results and qualitative factors that underlie the quantitative measurements. Section 6 compares this replication to the original experiment. Section 7 addresses methods, and Section 8 the validity and applicability of the experiment. Section 9 concludes.

2. The Original Experiment

This section gives an overview of the design of the original experiment.

2.1. Objectives and Hypotheses

"If you have a hammer, everything looks like a nail." Thus, having learned some design patterns, it may be tempting for a designer to use them even in situations where their complexity and application may not be warranted, and a simpler solution is available.

Prechelt et al. (2001) wished to test whether the use of some specific patterns in such situations is "helpful", "harmful" or "neutral" for subjects with different backgrounds. They informally framed their hypotheses as expectations: A design pattern P does, or does not, improve the performance of subjects doing maintenance work task X on program A (containing P) when compared with subjects doing the same work task X on an alternative program A' (not containing P).

Note that the programs may contain patterns other than the one being tested; these other patterns are used identically in the A and A' versions. Henceforth, we will use the term Pat for the version with patterns, and Alt for the version without.

The "helpful", "harmful" and "neutral" interpretations are derived from the support or contradiction of these hypotheses.

2.2 Variables

The experiment used three independent variables:

- *Program and work task:* There were four different programs, each with its own purpose, patterns, and two maintenance work tasks.

- *Program version:* Each program existed in two versions, functionally equivalent, comparably complex and sharing some code. The Pat version contained one pattern not present in the Alt version; the alternate version used a "simpler" structure to replace the pattern. This was the central variable of the experiment. Equivalent documentation was present in the two versions, such as an inheritance outline vs. pattern name and class role.

- *Amount of design pattern knowledge:* The experiment was divided into three parts. In the first half of day one, the subjects performed the pre-test, consisting of work tasks on two programs. The rest of day one and the first half of day two contained a patterns course, and the rest of day two was used for the remaining two tasks (post-test).

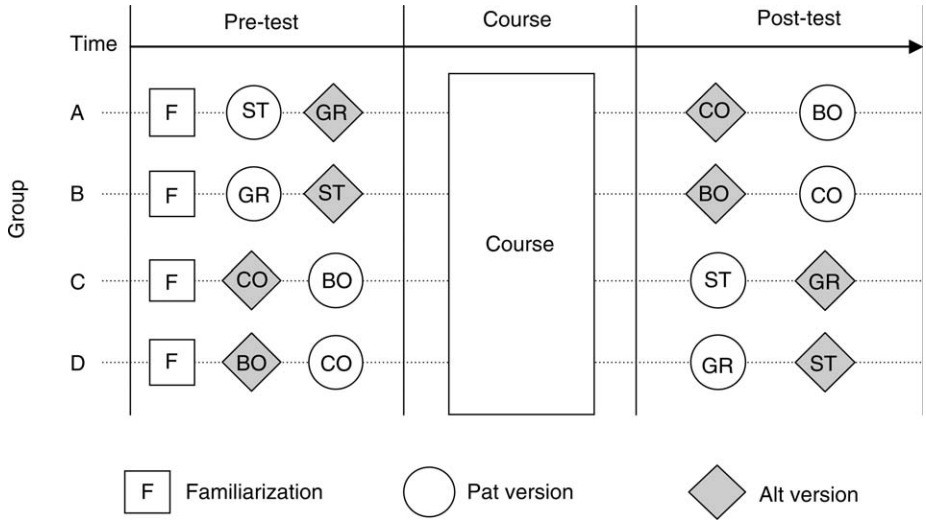

Figure 1. Experimental design: Circles denote Pat program versions, shaded diamonds Alt versions. The two-letter codes are the program name abbreviations. Time runs from left to right; the first day includes the pre-test and the first half of the course, the second day contains the second half of the course and the post-test.

Before the experiment, most subjects had little or no experience with patterns; thus, the post-test represented subjects with significantly more knowledge of design patterns than the pre-test. The experimental design is summarized in Figure 1. In order to control learning and fatigue effects, the order of programs is varied, and data are collected from each subject on both Pat and Alt programs.

The experiment used two dependent variables:

- *Time:* The time taken to complete each task, in minutes.

- *Correctness:* Each solution was evaluated on a five-point scale to assess to what degree it was functionally correct, regardless of whether it used the "proper" design. We use the term "correctness" instead of the more general "quality", as overall quality is complex and difficult to measure. However, a scale for "correctness" is more simply defined:

  1. *Requirements misunderstood*—the solution did not address the given task, or was totally useless, or no real solution was made or attempted.

  2. *Wrong answer*—the requirements were understood, but the attempted solution did not work and was not on the right track.

  3. *Right idea*—the requirements were understood and a reasonable solution attempt was made, but the solution either did not work or did not compile.

4. *Almost correct*—the solution compiled and ran but did not give exactly the correct answer; however, it did not contain any fundamental errors.

5. *Correct*—the solution compiled, ran and produced correct output.

2.3. Summary of Programs and Work Tasks

The four C++ programs used came from different domains and were of varying complexity. This was intended to guard against the possibility of domain knowledge or complexity systematically biasing the results; a fuller discussion can be found in Threats to Validity, Section 8.

This experiment looked at the effects of design patterns through the medium of code (with some documentation), not models. In a maintenance situation, there may not be any valid models available; also, having to actually implement changes in code provides a stricter test of understanding of the original code and its design. Fundamentally, the end result that matters from a development or maintenance project is the final code and not the underlying model.

Throughout this paper, the programs are identified by their names or abbreviations. The names reflect the domain of the programs: Stock Ticker (ST), Graphics Library (GR), Boolean Formulas (BO) and Communication Library (CO). For each program, there were two work tasks. With one exception, the first task was a programming task (addition of a feature), and the second task was oriented more towards theoretical comprehension.

Each program tested a different pattern. While it would be possible to create a program that contained all the patterns, it was considered easier to make the programs separately, so that each tested the aspects of a single pattern. The correspondence between the Pat and Alt versions of the programs was also considered simpler to maintain in separate programs. Most importantly, combining multiple patterns in a single program would have given the subjects a chance to see all of the patterns and code immediately, thereby introducing a learning effect and seriously compromising the experimental design.

In all cases, the features to be added to the programs corresponded to features already present in the code, which could be used as templates by the subjects. Table 1 contains short, comparable descriptions of all programs and tasks, while detailed descriptions can be found in Section 5.

2.4. Subjects, Programs, Tasks and Groups

A total of 29 subjects participated in the original experiment. They were all professional software engineers and came from a single company. Fifteen subjects had some prior experience with design patterns.

Table 1. Descriptions of programs and tasks.

Program	Description and complexity	Tasks	Patterns
Stock Ticker (ST)	Display an incoming stream of data (read from a file) in one or more windows using a supplied, simple GUI library. Simple program with little data and low code complexity. Pat: 441 SLOC, 7 classes Alt: 374 SLOC, 7 classes	1: Add another kind of window (the window itself was supplied). 2: Let the user choose which windows should be visible.	Pat: Observer Alt: None
Boolean Formulas (BO)	A system for storing and manipulating boolean formulas, represented in a hierarchical data structure. Relatively complex, using a recursive data structure. Pat: 471 SLOC, 11 classes Alt: 372 SLOC, 8 classes	1: Evaluation of formulas. 2: Change name of one method, breaking the Composite pattern.	Pat: Composite, visitor Alt: Composite
Comm. Library (CO)	Wrappers for communication primitives such as transmit, receive, compress and decompress. Very little data and not very complex. Simple primitives with similar interfaces. Pat: 404 SLOC, 6 classes Alt: 342 SLOC, 1 class	1: Add a wrapper for a new (supplied) primitive. 2: Determine the conditions leading to a certain status value; determine how to create a channel with certain functionality.	Pat: Decorator Alt: None
Graphics Library (GR)	Represent graphic primitives such as point, line and circle, and a system for drawing them on several kinds of device. Methods for creating devices and corresponding primitives. Data structure is partly recursive, but less complex than in Boolean Formulas. The code is larger than the other programs, but well structured. Pat: 683 SLOC, 13 classes Alt: 667 SLOC, 11 classes	1: Add a new graphics device and corresponding subclasses of primitives. 2: Determine whether a running supplied method will result in a certain output.	Pat: Abstract Factory, Composite Alt: AbstFactory

The subjects were divided into four groups (A–D). Each group maintained one Pat and one Alt version of a program in both the pre-test and post-test. Each subject worked on all four programs and each program was used as often in the pre-test as in the post-test, and as often in Pat and Alt versions, as shown in Figure 1.

2.5. Analysis and Statistical Methods

The time and correctness data were first evaluated using an analysis of variance to identify significant factors. As expected, the work task was the most significant factor, while the order of tasks was not significant. The rest of the factors (pre/post, pat/alt, individual differences) were discussed on a per-task basis.

Distribution-independent bootstrap methods were used to evaluate mean elapsed times and derive *P*-values for the differences (Efron and Tibshirani, 1993). Such differences were calculated for each pair that corresponded to a hypothesis, for example, for Pre-Alt vs. Post-Alt for a particular program and work task.

Numerous significant differences were found and compared with the expected trends (hypotheses) for each work task. For many tasks, all groups achieved near-perfect correctness, so the dependent variable "correctness" was often ignored (Prechelt et al., 2001, p. 1136).

3. Current Replication

We wished to increase the realism of the experiment (Sjøberg et al., 2002), and attempted to do so in two ways: (1) We used a real programming environment, instead of annotations to paper printouts; and (2) our subjects came from multiple consultancy companies and were paid for their participation, instead of being volunteers from a single company.

We used the same general design of the experiment and the exact same set of four tasks, with a Pat and Alt program version of each. The program code was identical to that used in the original experiment, except for corrections of minor errors. The same course on design patterns was taught by the same person (Walter Tichy), using the same course materials.

In the terminology of Lindsay and Ehrenberg (1993), this can be considered a relatively "close" replication. While an identical replication is neither possible nor particularly desirable, we designed ours to keep it as close as possible, except for differences that are either unavoidable or explicitly desired.

In this case, the difference in actual subjects and their nationality was unavoidable. Their background was roughly the same and was evaluated using the same methods. The use of paid subjects from more than one company, and the use of a programming environment, were motivated by the increased realism they offer.

3.1. Logging and Data Collection

Our subjects used their own laptop PC's as terminals, while the actual programming environment ran on a set of Windows Terminal Servers. This made it possible to install various non-intrusive logging tools to collect additional data, beyond the

correctness and time variables (as well as the post-mortem questionnaire) of the original design.

To gain insight into the programming process of each subject, a copy of the program file being edited was saved at every compilation, together with information on compilation errors, editing time (as a further check) and breakpoint and debugging information. It is, therefore, possible to detect the changes made for each compilation, which often occurs every few minutes, as well as the debugging method used. This data was used both for grading of solution correctness (Section 4.2) and in the qualitative analysis of results (Section 5).

To limit the possibilities for cheating, and to lessen competitive stress, subjects were placed so that two people sitting next to each other always worked on different programs. Copy/paste operations through the Terminal Server environment need multiple menu choices and intermediate files, making data exchange through IR ports impractical. Inspection of the code logs revealed no traces of cheating or plagiarism.

3.2. Subject Selection and Background

Several international and Norwegian consultancy companies contributed subjects. They were explicitly asked to provide people who formed a reasonably representative sample, with regard to seniority, experience and education.

In total, 44 subjects were paid for their participation, on three scales (junior, intermediate, senior). The employers determined the scale for each participant. Additionally, payment was offered for a limited amount of overhead per company, to encourage them to undertake a serious selection process.

The subjects were mostly (39) professional software engineers, from 11 different companies. There were also five students at the master/Ph.D. level. The median education was five years and work experience was four years. Five subjects had 20 or more years work experience. Seventeen subjects had some experience with patterns, though generally with only one or a few patterns, applied a few times. Only six subjects had practical knowledge of the patterns actually being tested.

Regarding prior experience with object-oriented programming and C++, one third of the subjects answered that they had less than one year experience with object-oriented programming (as opposed to other paradigms); the average value was 2.4 years. Seventy five percent of the subjects had written less than 25,000 lines of C++ code.

Thus, the participants in our replication generally had a relatively extensive education, but only limited practical experience, and initially almost no relevant pattern knowledge. We would expect the lack of practical experience to cause the subjects to spend more time on some programming details than would experienced developers. This also has some implications for the external validity of the experiment.

Table 2. Subject backgrounds for each of the four groups A–D.

	N	Pat	Educ	Work	OO prog	C++
A	10	1	3.5	5.7	2.2	13,300
B	12	1	4.7	5.3	2.6	16,387
C	12	2	4.0	6.7	1.5	8509
D	10	2	4.1	7.7	3.4	18,638

N = number of subjects; Pat = number with previous knowledge of relevant patterns; Educ = median education (years); Work = median work experience (years); OO prog = median OO programming experience (years); C++ = mean number of C++ lines of code written.

3.3. Group Assignment

The subjects were assigned to the four groups using randomized blocking, where the groups were balanced (the blocks were not random, but the assignment of members from each block to the groups was). Obviously, balancing all characteristics at once is not possible; the greatest weight was given to knowledge of design patterns, and general experience. The subjects completed a survey form before the experiment, and their answers were used to compute a ''pre-qualification score''. The subjects were ordered by this score, and those with the four highest scores were randomly assigned, one to each group, then the next four, etc.

Of the original 54 subjects who expressed an interest, 10 were unable to participate. Of those, four cancelled after the final group assignment, causing an imbalance in group sizes. Table 2 summarizes the groups. Possible threats to validity stemming from the imbalances are discussed in Section 7.8.

3.4. Experiment Conduct

The subjects were not told about the design of the experiment (the presence of Pattern and Alternate versions), nor about what we were measuring, logging or how this was done. The programming tasks were presented as exercises for the patterns course, though the subjects were told in advance that they were taking part in a combination of course and experiment. The authors discreetly eavesdropped on conversations during lunch and in breaks, and the subjects did not to our knowledge discuss the tasks.

The same general timetable was followed as in the original experiment: The pre-test work tasks in the morning, then lunch, followed by the first part of the design patterns course. On day two, the course continued until lunch, and the post-test work tasks were conducted after lunch.

The participants were encouraged to work until they were done. On day one there was a time limit (the start of the course), on day two there was no formal time limit and the last subject left at 5.45 pm. Four subjects ran out of time on day one, and in the questionnaire estimated that they would have needed from 1 to 3 hours

additional time to complete their tasks. Since our analysis looks at both the time needed for the tasks as well as the solution quality, these partial solutions were graded and included in the data set. The analysis of elapsed time excluded solutions of low quality, regardless of the reason for the low quality (Section 3.6).

3.5. Expectations and Hypotheses

Since both the programs and the different patterns they contained were of varying kind and complexity, the hypotheses varied. In some cases, we expected the Pat version to be easier to understand and modify, while in other cases we expected the Alt version to have the advantage. The expected effect of the patterns course also differed.

The hypotheses represent what we expected to observe based on software engineering common sense. They were identical to those of the original experiment, which used bootstrap methods to compare mean work times for work tasks, either between Pat and Alt versions or between Pre and Post. The hypotheses were defined and evaluated separately for each program and work task. We reformulated the hypotheses to correspond to our statistical approach, and they are presented in tabular form in Section 3.8, Table 3.

In addition to quantitative analysis of dependent variables, a qualitative analysis was also made, the purpose of the latter being to try to explain why the quantitative results were observed.

3.6. Model for Analysis of Time

To evaluate the observed quantitative data and enable a more compact representation of the hypotheses, a regression-based approach was adopted. The method used in the original experiment (bootstrap estimations of distributions of differences of means) only takes into account data for each pair of tasks considered, separately from all other data. The model adopted here considers all the data simultaneously and thereby enables us to better take into account differences between individual subjects.

Since completion times have little meaning for solutions with low correctness, only those solutions achieving correctness score 4 (“almost correct”) or 5 (“correct”) were used in this analysis.

The time used to execute a task may vary systematically by explanatory variables such as program and task number, Alt or Pat version, and amount of pattern knowledge. Define

$\text{time}_{t,i} =$ time used by individual i $(i = 1, \ldots, n)$ on task t $(t = 1, \ldots, 8)$, on the condition that the corresponding solution correctness was at least 4,
$I_{P,t,i} = 1$ if task t for individual i were with Pat, else $I_{P,t,i} = 0$,
$I_{C,t,i} = 1$ if task t for individual i were done after the course, else $I_{C,t,i} = 0$.

Further, let $E(\text{time}_{t,i}) = \mu_{t,i}$ be the expected time used on task t, where the expectation is taken over the sample population of programmers, given specific values of the explanatory variables. We assume that the logarithm of $\mu_{t,i}$ has the additive structure

$$\log(\mu_{t,i}) = \alpha_t + \beta_t I_{P,t,i} + \delta_t(1 - I_{P,t,i})I_{C,t,i} + \gamma_t I_{P,t,i} I_{C,t,i} \quad (1)$$

where the αs, βs, δs and γs are regression coefficients that will be estimated from the data.

The model can be transformed back to the original scale. The population averaged expected time used on task t before the course, for Alt programs, and with correctness at least four will be $\mu_{t,i} = b_t = \exp(\alpha_t)$.

The quantity b_t will be called the base level for task t. The expected time for a Pat program, before the course, and with correctness at least four, will be $\mu_{t,i} = b_t \cdot \exp(\beta_t)$, such that

$\exp(\beta_t)$ is the relative increase in time by using Pat instead of Alt before the course, that is, the "effect of design patterns before course".

Furthermore, the following quantities will be of interest:

$\exp(\delta_t)$	is the relative increase in time from Pre-Alt to Post-Alt, that is, the "course effect on alternate programs".
$\exp(\gamma_t)$	is the relative increase in time from Pre-Pat to Post-Pat, that is, the "course effect on design patterns programs".
$\exp(\beta_t + \gamma_t - \delta_t)$	is the relative increase in time by using Pat instead of Alt after the course, that is, the "effect of design patterns after course".

The relative increases will be reported as percentage increases, that is, instead of reporting $\exp(\beta_t)$, we will report $100 \cdot \exp(\beta_t) - 100$, etc.

If we assume that the observations $\text{time}_{t,i}$ are Gamma distributed, and independent for all t and i, the parameters can be estimated by maximum likelihood according to the theory of generalized linear models (GLM) (McCullagh and Nelder, 1989). The gamma distribution is suitable for data that takes only positive values and are skewed to the right. This is the case for the time data, which has 0 as lower limit, but no clear upper limit (though it cannot be longer than a day). However, the independence assumption is unrealistic, as we have multiple observations for each individual subject, one for each work task.

Therefore, the parameters were instead estimated by the method of generalized estimating equations (GEE) (Diggle et al., 1994; Liang and Zeger, 1986), using the software package Oswald (Smith et al., 1996). GEE is an extension of GLM, developed specifically to accommodate data that is correlated within clusters (here individuals).

First, the user has to specify a so-called working correlation matrix, i.e., the structure of the correlations between observations within the same individuals. For the present model, we have used an "exchangeable correlation matrix", which means

that all observations within the same individual have equal correlation. Then the estimation is carried out under the assumption that (the structure of) the working correlation matrix is true, and standard errors of the estimates are calculated.

The theory of GEE states that the estimates are asymptotically normal distributed with the given standard errors. Further, under certain assumptions, the estimates are consistent (i.e., converge to the true values when the number of observations becomes large), even if the distribution or the working correlation matrix is incorrectly specified. This important result does not imply that the choices of distribution and working correlation matrix are of no consequence. The closer they are to reality, the more precise the estimates will be.

3.7. Model for Analysis of Correctness

It would be natural to handle the correctness scores by ordinal logistic regression, i.e., by estimating the probabilities of getting the score values 1, 2, ..., 5, given the explanatory variables. However, it was impossible to estimate such a model by GEE or GLM, because the methods break down when all observations for certain combinations of the explanatory variables have the same value. This happened in several cases; for example, in the Pat group working on task 1 of the ST program in the post-test, all the subjects had a perfect score. Instead, we have used the model presented below, assuming Gaussian data.

The model for the quality or correctness score on each task is similar to that for time. Define $\text{score}_{t,i} \in (1, 2, 3, 4, 5) =$ score achieved by individual i $(i = 1, \ldots, n)$ on task t $(t = 1, \ldots, 8)$.

Further, let $E(\text{score}_{t,i}) = \mu_{t,i}$ be the expected score on task t, which is assumed to have the structure

$$\log(\mu_{t,i}) = \alpha_t + \beta_t I_{P,t,i} + \delta_t (1 - I_{P,t,i}) I_{C,t,i} + \gamma_t I_{P,t,i} I_{C,t,i} \tag{2}$$

The regression coefficients have different values than in the time model, and slightly different interpretations. The expected score for task t before the course, for Alt programs now becomes $\mu_{t,i} = \alpha_t$, where α_t will be called the base level for task t. The other coefficients give the increase in score compared with the same alternatives as in the time model. Note that positive values here mean improvements in correctness (higher correctness score), whereas negative values meant improvements in the time model (shorter time).

The parameters have again been estimated by GEE, but now using the Gaussian family of distribution as mentioned above. We used an "identity working correlation matrix", because the GEE algorithm did not converge with an exchangeable working correlation matrix. Using an identity working correlation matrix gives the same estimates as GLM, but the estimated uncertainty limits are more robust to incorrect specification of the correlation.

In practice, the correctness scores take integer values, and are far from Gaussian. As mentioned in the discussion of the time model, the GEE estimates are robust also

Table 3. Hypotheses for time.

Pr	Task	Post-Alt vs. Pre-Alt	Post-Pat vs. Pre-Pat	Pre-Pat vs. Pre-Alt	Post-Pat vs. Post-Alt
ST	1			S1 +	S2 –
	2				S3 –
BO	1	B2a –	B2b –	B1 +	B3 –
	2			B4 +	B6 0
CO	1			C1 –	C2 –
	2			C3a +	C3b +
GR	1	G2a –	G2b –	G1 +	
	2	G4a –	G4b –	G3a 0	G3b 0

to mis-specification of the distribution. However, more data would be necessary for the asymptotics to hold, so the uncertainty limits should be interpreted with some care.

To guard against a model optimized to find only the "desired" results and ensure its statistical correctness, it was constructed by one of the authors (M.A.) without prior detailed knowledge of the hypotheses posed.

3.8. Reformulated Hypotheses

Given the analysis models and the quantities $\exp(\beta_t), \exp(\gamma_t), \exp(\beta_t + \gamma_t - \delta_t)$, we can now express the hypotheses formally, in a tabular format. In Tables 3 and 4, a "+" in a cell means that we expected a positive value for the coefficient on the log scale or greater than 1 on the original scale (longer time, higher correctness score). A " – " means we expected a negative coefficient on the log scale or lower than 1 on the original scale. A "0" means we expected no change relative to the base level (log scale coefficient 0, original scale 1), which is the Alt version of each program, before the patterns course. The hypotheses and expectations are discussed in detail in Section 5.

Table 4. Hypotheses for correctness.

Pr	Task	Post-Alt vs. Pre-Alt	Post-Pat vs. Pre-Pat	Pre-Pat vs. Pre-Alt	Post-Pat vs. Post-Alt
ST	1				
	2				
BO	1				
	2			B5 –	
CO	1				
	2			C4a –	C4b –
GR	1				
	2				

Note that empty cells in this table mean that no hypothesis was advanced with respect to this program/task/parameter combination. As this is a replication of an earlier experiment, we did not change any hypotheses or advance any new ones. Significant observations that do not correspond to one of the hypotheses are discussed in Section 5.6. The numbering of the hypotheses was rendered consistent with those in the original experiment, to make comparisons easier.

The headings refer to the effect measured: Post-Alt vs. Pre-Alt shows the course effect (from pre-test to post-test) on the Alt version programs, while Pre-Pat vs. Pre-Alt shows the effect of going from Alt to Pat version, both taken in the pre-test only.

4. Results

4.1. Validation of Raw Data

The first step in the analysis was to check that there were no errors in the raw data resulting from misunderstandings or gross technical problems. The only such case was one subject who had performed the work tasks completely out of order. All data from this subject were therefore dropped.

4.2. Grading of Correctness

The grading of the solution correctness used the scale described in Section 2.2 above, ranging from "misunderstood" to "correct". Correctness was determined by first compiling and running the final solution saved by each subject. Then, the final solution code was inspected to determine the magnitude of any problems. Finally, all intermediate source files were inspected to arrive at a better understanding of any errors and the solution strategy. The grading was done by one of the authors using a system that presented the source files, program output, etc. The subject information was fully anonymized at this point (to the grader) and the subjects were graded in random order.

We also determined whether each solution used the patterns present in the code. Note that "correct" does not imply that the patterns present in the code, if any, were actually used in the solution; only that the solution produced the correct output.

Four subjects had consistently low-quality solutions. Inspection on a per-compilation basis revealed that their C++ proficiency was so low that it would significantly mask any other effect. None of them finished all tasks, and most had given up (i.e., stopped work while the solution was nonworking and there was more time available) on more than one task. All data from these subjects were also dropped. Since their solution correctness was consistently low across both Pat and Alt program versions, this introduces no significant bias.

4.3. Refinement of the Analysis Model

There were several candidates for explanatory variables other than those described in Sections 3.6 and 3.7. In the model for dependent variable time, candidate explanatory variables were the pre-qualification score and the correctness of the solutions. The pre-qualification score (as discussed in Section 3.3) should be significant if it is correlated with the actual performance of the subjects. An analysis of the data showed that this was not the case. The coefficient had a value close to 0 and was not significant.

We interpret this as showing that the pre-qualification score bears little relation to the subjects' actual performance. At the same time, the experimental design is quite robust with respect to effects of individual performance differences, and therefore also balancing of groups. Since all the subjects performed tasks on all the programs, imbalances between subjects and groups will increase the total variability, but have a relatively small chance of causing systematic skewing of the results.

Solution correctness could also have been included in the analysis model for time, in the form of an indicator variable $I_{Q,t,i}$ with value 1 if solution correctness 5 were achieved by a subject on a task. One might expect a positive value for this coefficient, indicating that achieving higher correctness takes more time. An analysis showed that it was also close to 0 and not significant.

Our interpretation is that high correctness was not achieved at the expense of time; i.e., skilled individuals tend to favor time and correctness equally.

In the model for dependent variable correctness, the pre-qualification score had a low, positive value ($p = 0.026$), but the values and confidence intervals of the other estimated coefficients in the model were not significantly changed by including this factor.

To summarize, including these candidate explanatory variables caused only very slight changes to the values and confidence intervals of the remaining coefficients, in the models for both time and correctness. They were therefore not included in the final model for time.

4.4. Effect of Programming Tool Use

We wished to determine whether any subjects had spent a significant amount of time on "technical details", that is, problems with programming language syntax, obscure compiler error messages or other factors that might be classed as not relevant to the effects of design patterns. This determination was necessarily exploratory in nature and proceeded as follows:

An analysis was performed of each separate compilation of each solution. A "syntactical change" was defined as one that did not introduce new features or functions, but only changed the statement that caused the compilation errors. Typically this consisted of trying out various combinations of the `.`, `->`, `::` or `*` operators, different placements of `[]` brackets in attempted array declarations, etc.

Another example was a subject who spent about 15 min looking for a missing closing brace; the error messages from the compiler were not helpful.

If the program submitted for compilation did not compile, the only changes were localized and syntactical, and there was a contiguous series of such changes all related to one or a few lines, those individual compilations were classified as "irrelevant technical detail". The sum of the editing times for such compilations was subtracted from the total elapsed time for that task, as a correction.

Such corrections[1] were made for 33 out of the 43 subjects. The regression analysis was then run on both corrected and uncorrected data, to check whether the corrections actually had any effect, and to guard against the introduction of any bias. Ideally, we would want the confidence intervals to shrink when using the corrections, though without significantly changing the point estimates.

The corrections did achieve some reduction in the confidence intervals, and did so without materially affecting the point estimates. However, the reduction was nowhere near significant and did not change the degree of support or rejection for any hypothesis. Since the grading that underlies such corrections is necessarily somewhat subjective, and there is a risk of penalizing subjects who simply spent time thinking about the problems without submitting compilations, the corrections were dropped and the final analysis done on uncorrected, raw data.

4.5. Summary of Quantitative Results

The results from the analysis using the regression models are shown in graphically in Figures 2 and 3, and in tabular form in Table 7 using the same layout as for the hypotheses in Table 3.

In the figures, the point estimates for the coefficients are dots and 95% confidence limits are shown as vertical bars; a significant (at 5%) result is one where the bars do not cross the 0 line. As detailed in Section 3.6, the estimates from the regression model are asymptotically normal distributed, providing the basis for calculation of the confidence intervals.

Descriptive statistics are given in Table 5 for working times and Table 6 for correctness scores. The first four columns in the tables contain the key to the measurement—the program, work task, Pat/Alt version, and pre-test/post-test.

From Table 7, we can see that significant results (at 5%) were achieved for five out of the total 20 hypotheses, while another seven tests showed a reasonably certain direction, either supporting or contradicting the hypothesis.

The regression model provides strong support for the hypotheses in four cases:

1. Using the Visitor pattern causes problems if the underlying data structure changes.

2. Decorator is a pattern that requires training, but then yields easier maintenance.

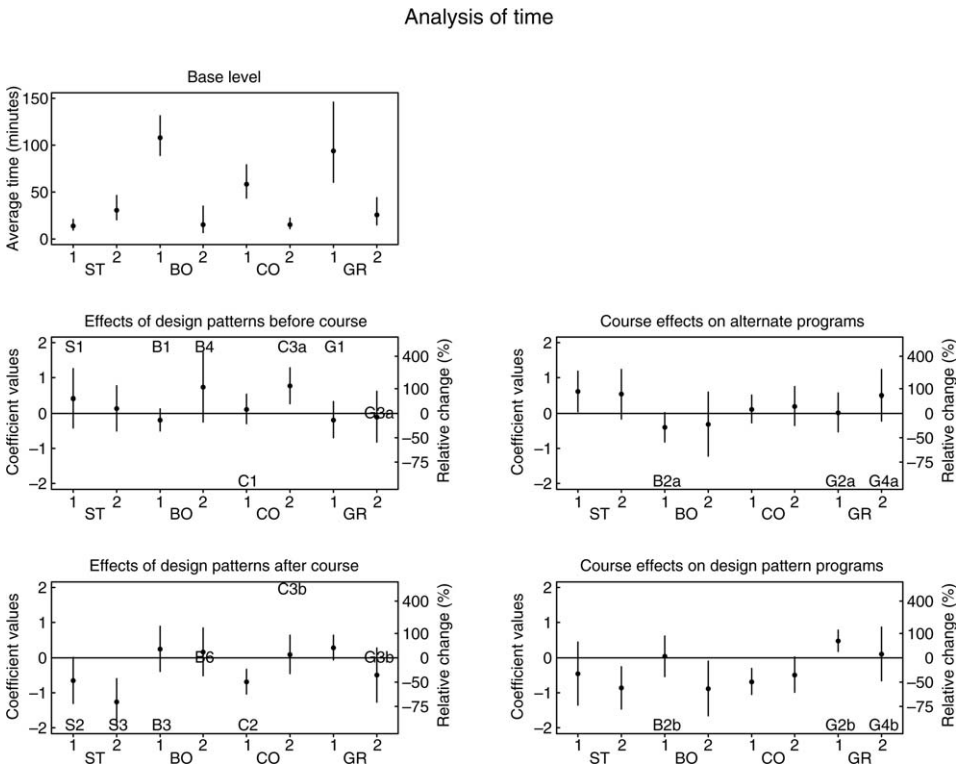

Figure 2. Analysis of elapsed times for all programs and tasks. The upper left panel of the figure shows the base levels b_t for each task, given in minutes. The estimates are given as dots, whereas the vertical lines are 95% confidence intervals. The four lower panels have the same structure as the upper left panel, but show changes relative to the base level. Hypotheses are shown, with the position of the label indicating the direction expected effect. The log scale is given at the left of the panel, and the corresponding relative change in % is given at the right side of the panel.

3. Decorator makes it more difficult to trace the flow of control in a program, and increases the time needed to understand it.

4. Like Decorator, Observer requires some training, but is then easy to understand and shortens maintenance.

The hypothesis that a short course is sufficient to profit from Abstract Factory was strongly contradicted. The observed result was actually the opposite; subjects took significantly longer after the course than before to complete the task.

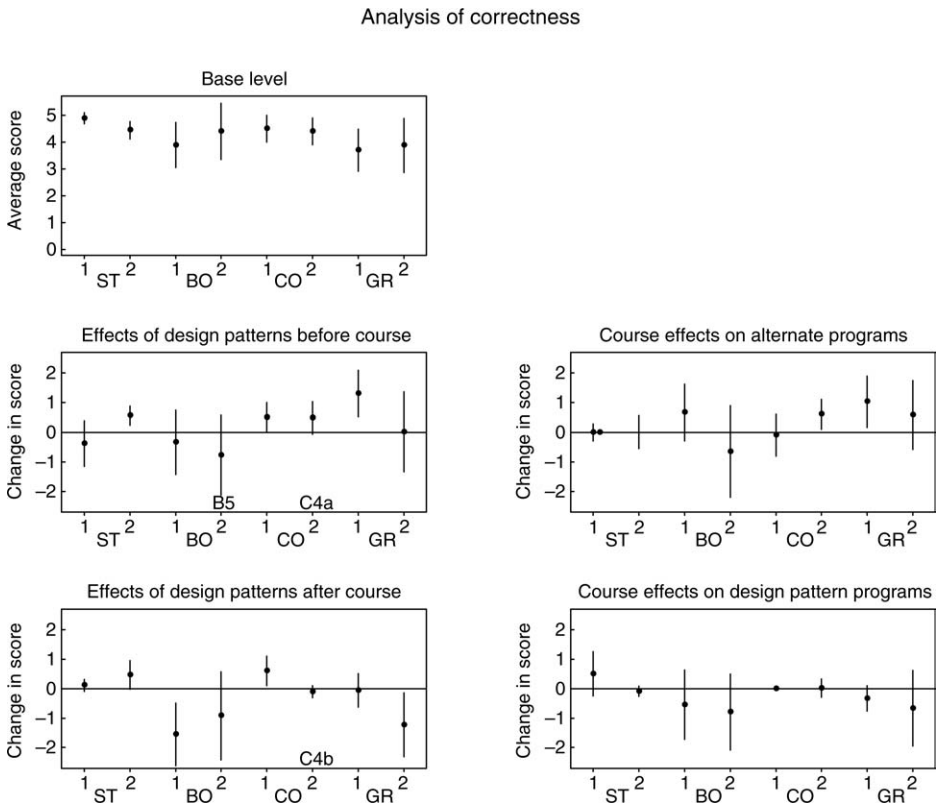

Figure 3. Analysis of correctness effects for all programs and tasks. The upper left panel of the figure shows the base levels α_t for each task, given as average scores. The four lower panels show changes in scores compared with the relevant alternative. Some confidence intervals have zero length. For the corresponding estimates, all relevant data had the same correctness, or the same change in correctness. In such cases, the GEE method is unable to compute confidence intervals.

5. Discussion

The quantitative results and hypothesis tests form only one part of the total results from the experiment, and do not automatically result in better understanding. The quantitative measures must be complemented with qualitative evaluations. Logging all compilations provides a basis for such evaluations, and allows us to gain insight into what happened, and from there to formulate explanations of why.

This discussion is structured around each program and work task, since their kinds and patterns were different, and revealed different aspects of patterns and their effects.

Table 5. Descriptive statistics for time: Each line contains the minimum, first quartile, mean, third quartile and maximum values of programming time in minutes, for one combination of program, Alt or Pat version, task number and day number. Number of subjects is also given. Each hypothesis in Table 3 refers to one pair of lines in this table, e.g., hypothesis (B2a: −) compares BO/Alt/Task 1/Post: line 2 to BO/Alt/Task 1/Pre: line 1, and expects the former to be lower (shorter time).

Line	Prog	Task	Pre/post	Ver	Min	$Q1$	Mean	$Q3$	Max	N
1	BO	1	Pre	Alt	27	88	129	175	298	9
2	BO	1	Post	Alt	27	45	86	130	186	9
3	BO	2	Pre	Alt	0	4	13	20	37	6
4	BO	2	Post	Alt	4	8	14	17	39	8
5	BO	1	Pre	Pat	45	86	108	135	145	11
6	BO	1	Post	Pat	20	55	99	160	173	10
7	BO	2	Pre	Pat	6	13	26	39	66	8
8	BO	2	Post	Pat	1	3	24	46	65	6
9	CO	1	Pre	Alt	22	35	63	85	97	10
10	CO	1	Post	Alt	24	48	69	90	117	10
11	CO	2	Pre	Alt	7	9	15	20	33	10
12	CO	2	Post	Alt	7	9	19	27	47	10
13	CO	1	Pre	Pat	44	45	65	84	124	8
14	CO	1	Post	Pat	17	22	33	43	60	9
15	CO	2	Pre	Pat	12	18	33	45	66	8
16	CO	2	Post	Pat	7	11	20	29	45	9
17	GR	1	Pre	Alt	32	51	91	136	196	10
18	GR	1	Post	Alt	37	57	105	155	214	11
19	GR	2	Pre	Alt	13	16	37	51	125	8
20	GR	2	Post	Alt	11	22	41	43	122	11
21	GR	1	Pre	Pat	36	59	78	89	155	9
22	GR	1	Post	Pat	101	104	127	145	190	9
23	GR	2	Pre	Pat	9	14	28	42	65	9
24	GR	2	Post	Pat	6	9	24	38	50	9
25	ST	1	Pre	Alt	6	7	14	21	32	9
26	ST	1	Post	Alt	7	13	26	35	65	9
27	ST	2	Pre	Alt	12	15	31	48	68	9
28	ST	2	Post	Alt	2	23	50	77	143	9
29	ST	1	Pre	Pat	1	3	22	27	85	10
30	ST	1	Post	Pat	2	6	14	26	40	11
31	ST	2	Pre	Pat	1	14	36	56	91	9
32	ST	2	Post	Pat	5	7	15	22	36	11

5.1. Observer: Stock Ticker (ST)

ST is a program for directing continuous streams of data (stock trades) to one or more displays that are part of the program.

Both versions of ST consist of seven classes, and in the Pat version (441 lines) four of them participate in an Observer. The Alt version includes one class that contains an instance variable for each display, and updates the displays when data changes.

Table 6. Descriptive statistics for quality: Each line contains the minimum, first quartile, mean, third quartile and maximum values of the correctness score, for one combination of program, Alt or Pat version, task number and day number. Number of subjects is also given. Each hypothesis in Table 4 refers to one pair of lines in this table.

Line	Prog	Task	Pre/post	Ver	Min	$Q1$	Mean	$Q3$	Max	N
1	BO	1	Pre	Alt	1	3.0	3.9	5.0	5	9
2	BO	1	Post	Alt	3	4.0	4.6	5.0	5	9
3	BO	2	Pre	Alt	2	2.8	4.2	5.0	5	6
4	BO	2	Post	Alt	1	2.0	3.8	5.0	5	8
5	BO	1	Pre	Pat	2	2.0	3.5	5.0	5	11
6	BO	1	Post	Pat	1	1.8	3.0	5.0	5	10
7	BO	2	Pre	Pat	2	2.3	3.6	5.0	5	8
8	BO	2	Post	Pat	2	2.0	2.8	4.3	5	6
9	CO	1	Pre	Alt	3	3.8	4.5	5.0	5	10
10	CO	1	Post	Alt	3	3.8	4.4	5.0	5	10
11	CO	2	Pre	Alt	3	3.8	4.4	5.0	5	10
12	CO	2	Post	Alt	5	5.0	5.0	5.0	5	10
13	CO	1	Pre	Pat	5	5.0	5.0	5.0	5	8
14	CO	1	Post	Pat	5	5.0	5.0	5.0	5	9
15	CO	2	Pre	Pat	4	5.0	4.9	5.0	5	8
16	CO	2	Post	Pat	4	5.0	4.9	5.0	5	9
17	GR	1	Pre	Alt	1	3.0	3.7	5.0	5	10
18	GR	1	Post	Alt	3	5.0	4.7	5.0	5	11
19	GR	2	Pre	Alt	2	2.0	3.9	5.0	5	8
20	GR	2	Post	Alt	2	4.0	4.5	5.0	5	11
21	GR	1	Pre	Pat	5	5.0	5.0	5.0	5	9
22	GR	1	Post	Pat	3	4.5	4.7	5.0	5	9
23	GR	2	Pre	Pat	2	2.0	3.9	5.0	5	9
24	GR	2	Post	Pat	2	2.0	3.2	5.0	5	9
25	ST	1	Pre	Alt	4	5.0	4.9	5.0	5	9
26	ST	1	Post	Alt	4	5.0	4.9	5.0	5	9
27	ST	2	Pre	Alt	4	4.0	4.4	5.0	5	9
28	ST	2	Post	Alt	3	4.0	4.4	5.0	5	9
29	ST	1	Pre	Pat	1	4.8	4.5	5.0	5	10
30	ST	1	Post	Pat	5	5.0	5.0	5.0	5	11
31	ST	2	Pre	Pat	5	5.0	5.0	5.0	5	9
32	ST	2	Post	Pat	4	5.0	4.9	5.0	5	11

There is no dynamic registration of observers in this version, which has 374 lines. The line counts include blank lines and comments.

The actual displays are implemented using a very simple `Window` interface. The subjects did not use this interface directly, because the display objects required by the work tasks were already present in the code.

Table 7. Summary of quantitative results—work time.

Pr	Task	Post-Alt vs. Pre-Alt			Post-Pat vs. Pre-Pat			Pre-Pat vs. Pre-Alt			Post-Pat vs. Post-Alt		
ST	1							S1	S	+52%	S2	**SS**	−48%
	2										S3	**SS**	−72%
BO	1	B2a	S	−33%	B2b	—		B1	WC	−17%	B3	WC	+29%
	2							B4	S	+108%	B6	WS	+18%
CO	1							C1	WC	+13%	C2	**SS**	−49%
	2							C3a	**SS**	+117%	C3b	—	
GR	1	G2a	WC	+2%	G2b	**SC**	+62%	G1	—				
	2	G4a	C	+66%	G4b	—		G3a	S	−9%	G3b	C	−39%

Cell contents: To the left is the program/hypothesis identification, followed by the degree of support in the center. SS means strongly supported; SC means strongly contradicted (significant at 5% level); S means supported; C means contradicted (not strictly significant at 5%, but still relatively clear effect); WC means weakly contradicted; similarly, WS denotes weak support; "—" denotes an inconclusive result. The estimated effect in % of the factor on the observed time (multiplicative) is given to the right.

5.1.1. *Work Task 1*

"In the given program, only one of the three display types is used. Enhance the program such that a second display (of type `volume`) is shown".

The Pat groups only had to invoke the pattern method `subscribe()` with a new instance of the display. The Alt groups had to introduce an instance variable for the new window and invoke its `update()` method to show new data. The main work in this program is to understand how it operates, because the actual changes required are very small in both cases.

Hypotheses: In the pretest, we expected the Pat group to need more time (S1: +), since subjects without pattern knowledge need to analyze the Observer to determine how it operates. After the course, we expected the opposite (S2: −), because the Observer should then have been easy to grasp and use.

Results: S1: $+ \rhd + 52\%^{+253}_{-34}$ is weakly supported.[2] Pre-Pat subjects did use 52% more time than Pre-Alt. Most of the difference can be attributed to one outlier data point; without it, the Pre-Pat and Pre-Alt groups are virtually identical with respect to time used. The correctness is also consistently high for both groups.

The outlier is a subject who did not understand the Observer pattern, and actually reinvented and reimplemented an equivalent structure, which explains why it took so long to complete the task.

Hypothesis $S2 : - \rhd - 48\%^{0}_{-73}$ is quite strongly supported. There were no significant differences in the correctness of the solutions.

5.1.2. Work Task 2

"Change the program so that displays can be dynamically selected at runtime." The code included a third display type for this purpose, as well as a simple method to get user input: `bool askYesNo( char *prompt );`

The Pat groups needed to do very little; just ask the user two or three questions, and `subscribe()` to those displays that were selected. The Alt groups would have to add a mechanism for extending the number of displays with more instance variables.

Hypotheses: In contrast to the original pen and paper experiment, where Prechelt et al. (2001) stated that the Pat groups did not need to do anything, the subjects in this experiment actually had to implement a change. We did, however, expect the Pat groups to have a clear advantage (S3: $-$), because their changes are much smaller; apart from getting user input, only a test around each `subscribe()` method invocation is needed.

Results: S3: $- \rhd - 72\%_{-85}^{-45}$ is strongly supported. The Post-Pat group spent less than half the time used by the Post-Alt group. Most subjects in the Post-Pat group chose to add Boolean variables to the `TradeInfo` class constructor to specify the users' choice, while a few put the subscribe calls into the main program.

All the subjects in the Alt groups interpreted the task to mean a choice between a fixed number of displays: those already present in the code. None implemented a fully dynamic solution that would easily have accommodated another display.

Correctness was significantly higher for the Pat groups than for the Alt groups both before and after the course. The effect of the course was in this case to reduce the time needed to arrive at a high-correctness solution. Observer, therefore, seems to be a pattern that can be grasped without much training, but training saves time.

5.2. *Composite and Visitor: Boolean Formulas (BO)*

Boolean Formulas contains a library for representing Boolean formulas (`OR`, `AND`, `XOR`, `NOT` and variables), and methods for printing the formulas in two different styles. It also contains a small main program that sets a few variables, constructs a formula and prints it using both methods.

The Pat version consists of 11 classes over 471 lines. The formulas are represented using a Composite, and the printing methods use Visitors. For each concrete Composite class there is a printing method in each of the Visitors, and each Composite class provides a dispatch method for the Visitor. Internally, the Composites use three different data structures: `NOT` has a single operand, `XOR` has two operands in a classic left-right scheme, while `AND` and `OR` are implemented with a common base class and have a dynamic number of operands to handle expressions such as `a AND b AND c` (this would be one `AND` with three operands). Recursion is a central feature of the Composite pattern. The Visitor solution allows the addition of new functions without changing the Composites.

The Alt version has the same Composites, but is shorter, with eight classes over 372 lines. The Visitor is completely missing, and the printing functionality is implemented directly as methods in each concrete Composite, so adding a new function means adding methods to each concrete Composite.

5.2.1. *Work Task 1*

"Enhance the program to evaluate Boolean formulas, i.e., to determine the result for a given formula represented by a Composite and values of the variables."

The printing methods serve as structural examples. The Pat groups had to create a new Visitor, while the Alt groups had to add new methods to each concrete Composite class.

Hypotheses: In principle, it should be easier to create a single new class similar to another existing class, rather than having to add methods to several classes. This should favor the Pat group.

However, Visitor is quite a difficult pattern to comprehend and use, so we expected that the Pat group would need more time to understand the structure than the Alt group would need to simply add methods (B1: +).

Gaining patterns knowledge during the course should help both groups, since there is a Composite in both the Pat and Alt versions of the program (B2a: −), (B2b: −). The Pat group should get an additional advantage from the Visitor pattern after the course (B3: −).

Results: B1: $+ \triangleright - 17\%^{+14}_{-40}$ was not supported; the Pre-Pat group actually needed 17% less time than the Pre-Alt group, though much of the difference was due to one outlier.

A more interesting observation comes from the correctness model in Table 8 (Visitor). First, the correctness was quite low, and lower for the Pat group before the course. Second, the Alt group benefited from the course, while the Pat group actually

Table 8. Summary of quantitative results—correctness.

Pr	Task	Post-Alt vs. Pre-Alt	Post-Pat vs. Pre-Pat	Pre-Pat vs. Pre-Alt			Post-Pat vs. Post-Alt		
ST	1								
	2								
BO	1								
	2			B5	S	−20%			
CO	1								
	2			C4a	C	+15%	C4b	WS	−5%
GR	1								
	2								

got worse. Perhaps most interesting is that only three out of 12 subjects in the Pre-Pat group actually used the visitor. The rest implemented changes directly on the composite and ignored the two Visitors in the code.

B2a: $-\triangleright-33\%^{+2}_{-56}$ and B2b : $-\triangleright+4\%^{+86}_{-42}$ had some support, most for B2a. There was, however, some rise in correctness for the Alt group.

B3: $-\triangleright+29\%^{+144}_{-32}$ was inconclusive, with no significant difference visible. However, the correctness of the Pat group solutions was at the same low level as in the pre-test, and the subjects were still not using Visitor much—four out of 10, and of those four, only one succeeded.

Inspection of the solutions on a compilation-by-compilation basis revealed that many subjects struggled with the recursion inherent in the Composite. This is somewhat surprising, given that the subjects were professional developers, many employed by major consultancy corporations.

The conclusion is that Visitor was so difficult that even after a course that gave the instructor excellent feedback (grade better than 4 out 5), most subjects either ignored the pattern or were confused by it.

We may also speculate that developers nowadays use predefined container classes so much that recursion is simply not used on a daily basis any more. This has implications for the design of future experiments, and for the usefulness of design patterns that depend heavily on recursion in their structure.

5.2.2. Work Task 2

"After a code review, an incompetent manager requires you to change the method `operatorname()` to `varname()` on the `VarTerm` class only".

This in effect broke the Composite pattern, because one of the concrete classes no longer followed the declaration of the superclass.

Hypotheses: We expected that the Alt group would find the pre-test easier than the Pat group (B4: +), because all the resulting changes only have to be made in the class already being changed. The Pat group would have to modify the Visitor classes. One of the modifications is somewhat tricky to spot, so we also expected lower correctness (B5: −).

In the post-test, we expected that the Pat and Alt groups would be roughly equal, because the magnitude of the change is the same in both cases (B6: 0).

Results: B4: $+\triangleright+108\%^{+456}_{-22}$ seemed to be supported, in that the Pat group needed twice as long as the Alt group. However, we were still dealing with subjects who in five cases out of seven were not using Visitor. There was quite a lot of confusion visible in the solutions, and the correctness was quite low B5: $-\triangleright-20\%^{+18}_{-40}$. Also, four subjects were missing (relative to work task 1), because they had run out of time.

B6: $0\triangleright 18\%^{+132}_{-40}$ was contradicted; the correctness of the Pat solutions was markedly lower than that of the Alt solutions (average grade 2.8 vs. 3.8).

This reinforces the conclusion from work task 1: our subjects were badly confused by the Visitor pattern; only 6 out of 21 achieved a "correct" or "almost correct" solution (Pre and Post combined).

5.3. Decorator: Communication Channels (CO)

This program is mostly a wrapper library. A communication channel establishes a connection for transparently transferring packets of data of arbitrary length. One can turn on additional functionality for logging, compression and encryption.

The library does not implement the functionality itself, but only provides a Façade for a system library (whose internal source code was unavailable to the subjects). However, this application of the Façade pattern is irrelevant to the experiment.

The Pat version uses a Decorator scheme to add extra functionality to a bare channel. It consists of six classes over 404 lines.

The Alt version has only a single class, using boolean flags and `if` sequences for turning functionality on and off during the processing of one packet. It consists of 342 lines, and is the only instance where the Alt version has a structured (as opposed to object-oriented) design.

5.3.1. Work Task 1

"Enhance the functionality of the program so that error correction can be added."

The actual encoder/decoder for the error correction was available as an interface with a working, hidden implementation. Its interface was exactly analogous to one of the functions that were already in use.

Hypotheses: Decorator has two competing influences. On the one hand, it nicely separates the functionality and all but removes dependencies, so that it should be easy to add a new function by adding a new Decorator, rather than having to find the right place in a block of `if` statements.

On the other hand, this separation also means that it is more difficult to trace what is actually going to happen at runtime, as opposed to a situation in which there is a structured block of code with a clear sequence of flag tests.

We expected the first influence to be stronger, and hence that it would be quicker to enhance the Pat version (C1: $-$), especially at higher levels of pattern knowledge (C2: $-$).

Results: C1: $- \rhd + 13\%^{+69}_{-25}$ was not supported, because the groups spent virtually the same time. However, the correctness of the solutions was much better for the Pat group, with a perfect (10/10) "correct" score for all Pre-Pat subjects. In the Pre-Alt group, there were nine "correct", one "almost correct" and two "right idea".

Expectation C2: $-\triangleright-49\%_{-64}^{-28}$ was strongly supported. Also the correctness was again better with nine "correct", vs. six "correct", two "almost correct" and two "right idea".

The conclusion here is that Decorator is a pattern that can be grasped with reasonable ease, and contributes to higher correctness. With training, its use also results in considerably faster development.

5.3.2. Work Task 2

A communication channel—as implemented in the program—has an internal state `(open, closed, failed)` that is altered by certain operations. Work task 2 asked the subjects to "determine when a `reset()` call will return the '`impossible`' result". This required the subjects to find where the underlying state was changed, and how. This should be easier for the Alt group, where the state changes are strongly localized, so we expected shorter time (C3a: +), (C3b: +) and higher correctness (C4a: −), (C4b: −).

The subjects were also asked to "create a channel that performs compression and encryption". Again, the Alt group should have the advantage, since they only needed one new statement. The Pat group needed to determine the correct nesting of Decorators to achieve the same result.

Results: C3a: $+\triangleright+117\%_{+31}^{+260}$ was strongly supported in the pre-test, but inconclusive in the post-test (C3b: $+\triangleright+9\%_{-37}^{+88}$). Correctness was actually better for the Pat group in the pre-test, contrary to C4a: $-\triangleright+15\%_{-5}^{+22}$, and remained so in the post-test C4b: $-\triangleright-5\%_{-10}^{+5}$.

This seems to reinforce our conclusion that Decorator had a mainly positive effect and can be grasped without too much training. Any problems caused by the delocalization that results from applying this pattern were outweighed by the greater ease of composition of functions.

5.4. Composite and Abstract Factory: Graphics Library (GR)

The Graphics Library enables creation, manipulation and drawing of simple graphical objects, such as points, lines and circles. They can be rendered to different displays (alphanumeric or pixel), represented as output device classes with standardized interfaces.

In a central class a device context (type) is selected, and depending on this choice different versions of the graphical objects are created. Some basic objects (points and lines) are implemented identically for all devices, but circles have special implementations per device. Objects can also be collected in groups, which can then be manipulated like objects themselves.

The Pat version uses AbstFactory for the generator classes, and composite for the hierarchical object grouping.

The Alt version uses a single generator class with `switch` statements for the different devices, per object type. Combination and manipulation of objects is achieved with a quasi-Composite, the only difference being that there is no hierarchical group nesting.

This pair of programs has the smallest structural differences of all of the four pairs. The Pat version has 13 classes over 683 lines, the Alt version uses 11 classes over 667 lines. Both versions contain an identical main program that defines an Olympic logo with five circles and a line, rotates it 180° and draws it.

Output devices are represented by working classes with simple interfaces and hidden implementations, and can be run by the subjects. Some sample output is also included in the task documentation.

5.4.1. Work Task 1

"Add a third device type (a pen plotter)." The Pat group had to introduce a new concrete factory class, extend the factory selector method, and add two concrete product classes using the supplied `Plotter` device interface. The Alt subjects instead had to extend the `switch` statements; the product classes were the same.

This task was also one in which there was a difference from the original experiment, because in the present case, subjects actually tried out their solutions and ran into issues with scaling, forgetting to select a visible pen, etc. These issues affect the "product classes", which are common to both the Pat and Alt versions. Since this extended the total time, we expected to see less difference overall than in the original experiment.

Hypotheses: The actual volume of changes is the same for the Pat and Alt groups, so the main difference should come from differences in comprehension. The Alt program, with its localized `switch` statements, should be easier to understand (G1: +).

Pattern knowledge should help both groups to deal with the composite (G2a: −), while the Pat group may derive an additional advantage from better understanding of the AbstFactory (G2b: −).

Results: G1: $+ \triangleright - 17\%^{+40}_{-50}$ was inconclusive. If we omit one Alt outlier, the groups were virtually identical with respect to time used.

However, the correctness was significantly better for the Pat group. Inspection of the code revealed that several of the Pre-Alt subjects did not use the supplied `Plotter` interface, but instead just copied one of the existing output device interfaces. However, this did not necessarily invalidate the test, since it is the structure more than the particular class that matters.

The post-test, G2a: $- \triangleright + 2\%^{+76}_{-41}$, was inconclusive, with a hint in the opposite direction—subjects needed more time after the course for the Alt version. G2b: $- \triangleright + 62\%^{+120}_{+19}$ was not supported: The Pat group needed more time than the Alt group; this time all the subjects actually implemented the plotter as intended.

It seems that use of AbstFactory therefore had little influence on the time needed to make the changes, but it may have contributed positively to quality. This agrees with the purpose of the pattern: to concentrate the knowledge of which objects should be created in one place. The actual work remains roughly the same, but is localised instead of being spread out.

5.4.2. Work Task 2

"Determine whether a given sequence of statements will result in an *x*-shaped figure." This is a comprehension test on Composite, where the key is to recognize that references, and not copies of objects, are stored in an object group.

Hypotheses: First of all, we expected the subjects to actually try running the function containing the statements (it is present in the program, but not called by default). This was in contrast to the original experiment, where analysis was the only possibility. Consequently, we expected correct answers.

The structure of both programs is similar, so no difference in time was expected for Pat and Alt (G3a: 0), (G3b: 0), but we did expect the post-test to be faster than the pre-test, due to knowledge about the Composite (G4a: –), (G4b: –).

Results: High correctness was present only for the Post-Alt group. All other groups had a significant number of incorrect answers, despite the fact that most of them tested their solutions (visible traces in the code logs).

G3a: $0 \triangleright -9\%^{+86}_{-56}$ and G3b: $0 \triangleright -39\%^{+32}_{-72}$ were inconclusive. The Post-Pat group spent 39% less time than the Post-Alt group, but their correctness was significantly lower. Perhaps they were tricked by the application of Composite and forgot to take the effect of object references vs. object copies into account.

G4a: $- \triangleright +66\%^{+242}_{-20}$ and G4b: $- \triangleright +11\%^{+139}_{-48}$ were not supported; there was no significant difference in time spent between the pre-test or post-test. If anything, G4a tended in the opposite direction, with subjects spending more time on the Alt version after the course. However, this may simply be due to fatigue.

Due to scaling, a part of the figure would have been clipped and invisible, so we conclude that a number of subjects tried the drawing method, and when they did not get a good visible result, they resorted to (faulty) analysis instead of getting a good test output. After that, we were probably measuring a combination of their C++ proficiency with pointers and their approach to testing, rather than knowledge of design patterns. However, the fact that they trusted their analysis rather than actually making the test run well is interesting in itself.

5.5. Summary of Qualitative Results

We found evidence for both the usefulness and potential for harm of using design patterns, mostly as predicted by software engineering common sense. In summary:

Observer—expectation: The pattern solution is more complicated and harmful relative to the alternative solution, unless its flexibility is required.

Actual result: There was no significant harmful effect, even for subjects with little or no patterns knowledge. After a short course, a significant benefit was observed in terms of both time and correctness.

Composite, Visitor—expectation: Visitor is difficult to understand and thus harmful.

Actual result: Both before and after the course, the majority of the subjects did not even use Visitor even though two examples were present in the code. The correctness of the solutions was significantly lower than in the alternate design.

Decorator—expectation: Delocalization of functionality is expected to make it easier to change, but more difficult to analyze and call.

Actual result: The first expectation was supported, but the second was contradicted. The correctness and time improved significantly after the course, but the correctness was also better in the Pat version before the course, for no large penalty in time spent.

Composite, AbstFactory—expectation: The similarities in the designs of Pat and Alt versions lead us to expect only minor differences.

Actual result: No overriding difference was observed, even though the course helped with the AbstFactory pattern, which was also present in the Alt version.

5.6. Other Observed Effects

In this section, we discuss cases where a significant effect was observed, but no prior hypothesis existed. This situation arises because of the regression-based analysis model, which automatically estimates more coefficients than those needed to test the hypotheses of the original experiment. These results are necessarily of an exploratory nature.

Since there are no hypotheses to use for labeling, we will instead refer to the panel title in Figure 2 or 3, together with program abbreviation and task number. The observations are grouped by program and thus pattern.

5.6.1. Stock Ticker/Observer

Figure 2 (time): Course effects on design pattern programs, work task 2—this panel compares the time spent on the work task before (pre-test) and after (post-test) the course, and there is a significant difference present. In the post-test, the subjects used 60% less time than in the pre-test.

In the corresponding coefficient for the Alt version, the opposite effect is present, though without being significant. So the effect of the course was to reduce the time used for the Pat version, and increase it for the Alt version.

Part of the explanation lies with two subjects; one spent some time looking for a typographical error (an extra closing brace that caused cryptic error messages from

the compiler), and the other struggled with the syntax of arrays and enumerators. These two outliers increased the time spent by the Alt group; but being outliers, did not cause the increase to be statistically significant.

That the course should benefit the Pat group is as expected. The nature of the program and task were such that understanding of the Observer pattern reduced the task to only a few lines. The corresponding panels in Figure 3 (correctness, course effects) show almost no discernible effects.

Figure 3 (correctness), effects of design patterns before course and effects of design patterns after course both point in the same direction: the solutions of the Pat version have a higher correctness. The effect is significant in the pre-test and close to significant in the post-test. It reinforces the conclusion in Section 5.5 and contradicts the expectation from the original experiment.

5.6.2. Communication Library/Decorator

Figure 2 (time): Course effects on design pattern programs, work task 1—the panel shows that the subjects who maintained the Pat version of the program spent significantly less time in the post-test than those who maintained it in the pre-test (50% less). In Figure 3, the panel effects of design patterns after course measures the difference between Alt and Pat version in the post-test, and shows a significant increase in correctness for the Pat version.

There is one outlier in the Pre-Alt group, who worked for about 90 min before submitting a large change for compilation. However, the measured effect persists even if this outlier is omitted. The tentative interpretation is that Decorator is a pattern that benefits significantly those who take even a short course in it, and that such benefit influences both the correctness of the solutions and the time used to complete them.

5.6.3. Base Levels

The size and complexity of the programs varied, and this can be seen in the "base level" (time used for the Alt version in the pre-test, correctness at least 4 out of 5, as defined in Sections 3.6 and 3.7). Qualitative analysis of the solutions showed that recursion and recursive data structures created problems for a number of subjects. This affected the BO and GR programs (longer time, lower correctness), but is not a threat to validity because recursion is a central feature of the Composite pattern that is present in these programs.

The ST program (Observer pattern) was the shortest and simplest, and the base levels show a short time and high correctness for the solutions. Again, this reflects the pattern itself, which is structurally very simple.[3] In this experiment the subjects had few problems understanding and extending it, and the base level observations reinforce the conclusions regarding this pattern.

6. Comparison with the Original Experiment

This experiment was originally designed to investigate whether it is useful to utilize design patterns during program construction, even if the particular problem can be solved in simpler ways. The context was program maintenance by developers other than the original ones, and the experiment used pen and paper.

Our replication of the original experiment added the dimension of a real programming environment, so we retained the original aim, while adding an interest in the effects of the differences in execution of the experiment. We therefore re-analyzed the data from the original experiment using the same regression model, estimation method and software as for our replication, thus making it easier to compare the two experiments.

The program CO had three work tasks in the original experiment. The last two were quite similar and had similar hypotheses; in our replication they were combined to give a more symmetrical experimental design. When re-evaluating the data from the original experiment, the completion times for the last two tasks of this program were summed, and the correctness scores averaged and rounded to the nearest integer to match the analysis model. In the original experiment, so many subjects misunderstood BO Task Two that it was omitted from the analysis. We likewise omitted it from our re-evaluation.

6.1. Base Level and Variance

For the dependent variable time, the upper left panel of Figure 4 shows that the trends are similar, in that the same work tasks take a long or short time to complete. However, the absolute values differ. Those tasks that contained a lot of programming (BO task 1, GR task 1) took significantly longer in the replication.

Our explanation is that this is mainly an effect of the switch to actual programming; since all the details have to be correct, there is more work to be done than simply sketching out a class on paper. The variances are also larger, and we attribute that to the same effect—Prechelt (2002) has previously estimated the expected speed difference between fast and slow engineers to be on the order of 4:1, and we expect both the programming environment and the varied background of the subjects to contribute.

Another possible contributing factor is the fact that our subjects were paid for participating, whereas in the original experiment, the subjects were volunteers. We would expect this to cause a greater variance, because the volunteers would generally be more interested than a less self-selecting sample. However, the presence or strength of this factor is difficult to evaluate separately.

For the dependent variable correctness, seen in the upper left panel of Figure 5, the situation is similar, though the correctness of the solutions in our replication is often somewhat lower than in the original experiment. One possible explanation is that the criteria for scoring were not identical. For instance, it is possible to grade as

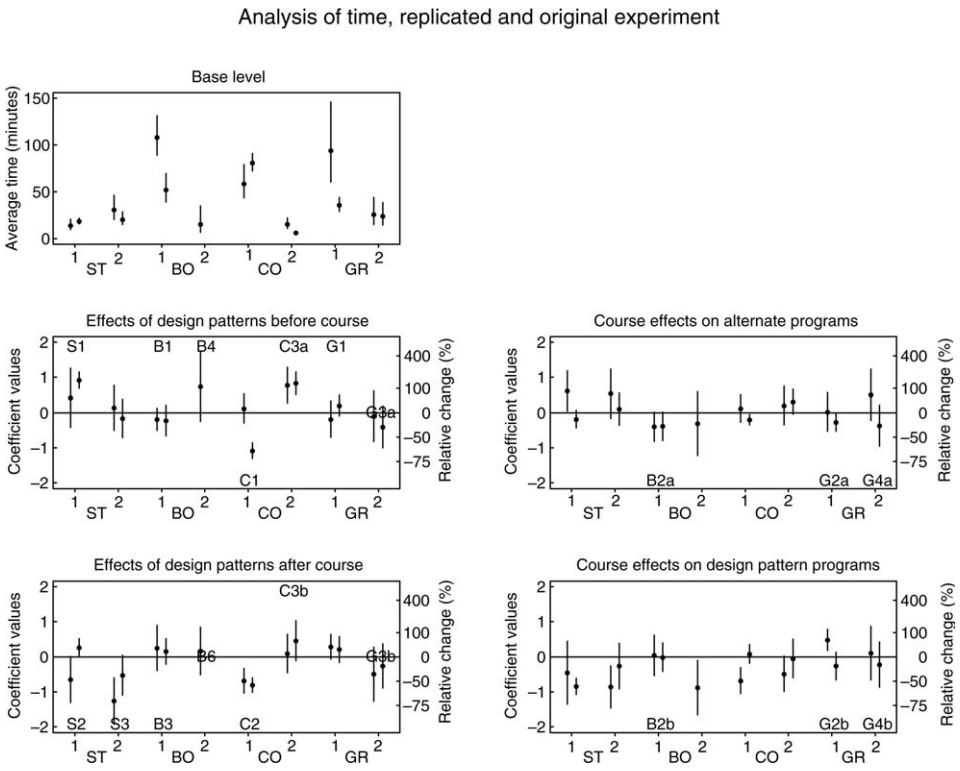

Figure 4. Completion times for all programs and tasks. Same format as Figure 2. For each coefficient, the left value is from the current replication, and the right value is from the original experiment, using the same analysis model and estimation method.

"correct" a paper solution that does not actually compile due to a syntax error; in our replication this situation would have given an "almost correct" score at best.

6.2. Elapsed Time

The four lower panels of Figure 4 show that in most cases, the sign, and to a lesser extent the size of the observed effects, are similar in both the original and the replicated experiments. Since the replication is fairly close, we concentrate the following discussion on the following cases: a hypothesis exists, its estimated actual coefficient changes sign and there is little or no overlap of the confidence limits. We first note that there seem to be no systematic differences. As the replication is fairly close, this is as expected, and improves our confidence in the validity of the experiment.

For ST/Observer work task 1, the hypothesis (S2: −) is contradicted in the original experiment and confirmed in the replication. The expectation was that given knowledge of design patterns, subjects would find it easier to implement the work

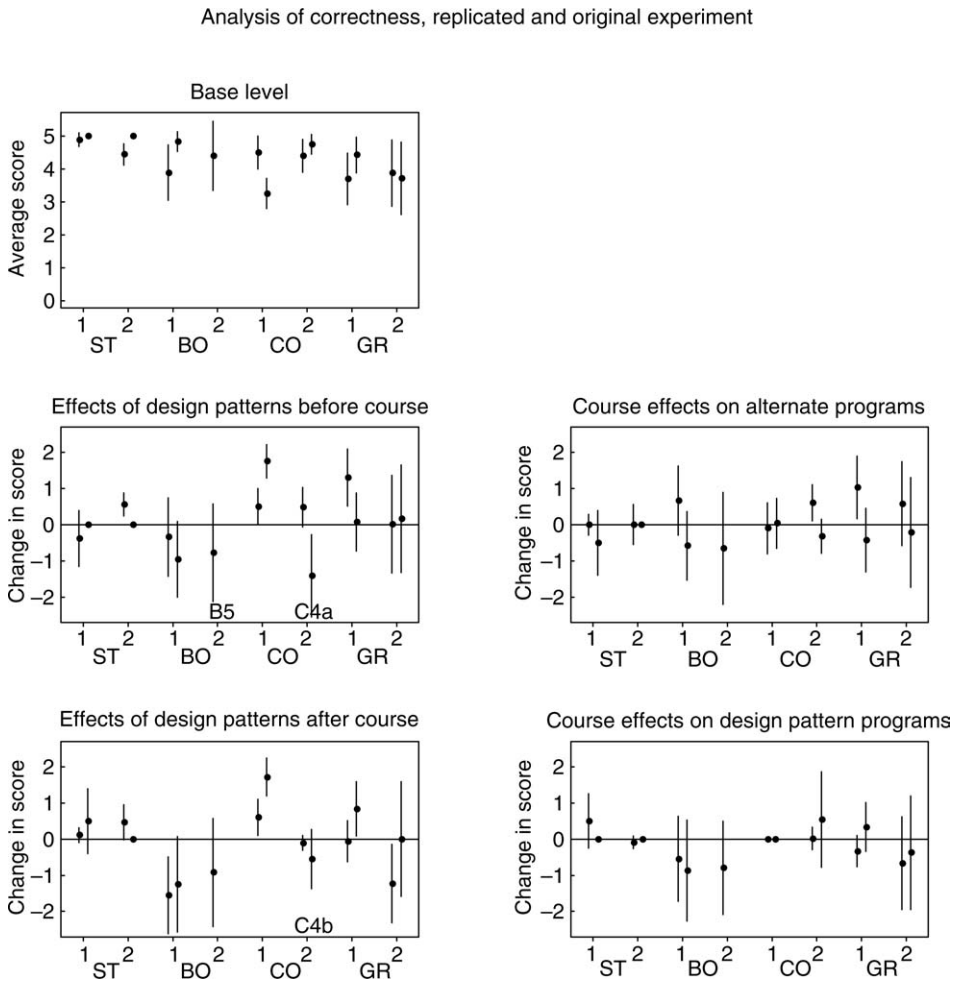

Figure 5. Correctness effects for all programs and tasks. Same format as Figure 3.

task on the Pat version, because the programming effort would be much smaller. This did not happen in the original experiment, while the expectation was supported in our replication, leading to opposite conclusions regarding the Observer pattern.

In CO/Decorator work task 1, there are conflicting expectations. The Alt version has a greater amount of localized code and should be easier to understand, but new functionality has to be added in several places and correctly sequenced, leading Prechelt et al. to expect the subjects working on the Pat version to be faster, even in the pre-test. In the original experiment, the first effect was the strongest, leading to a strong confirmation of (C1: −). In the replication, subjects working on the Alt version were marginally slower than those working on the pat version, leading to an inconclusive result. Simultaneously, correctness was significantly higher for the Pat version than for the Alt version in the replication, and even more so in the original

experiment. In summary, the subjects in the replicated experiment had more problems with Decorator before training than in the original experiment. In the post-test there is no significant difference between the two experiments.

6.3. Correctness

As with the dependent variable time, we see no large, systematic differences between the original experiment and our replication in the estimated coefficients for the dependent variable correctness. For CO/Decorator work task 1, we observed a smaller improvement in correctness going from the Alt version to the Pat version of the program than in the original experiment. The explanation lies in the base level correctness (Alt version), which is higher for the replicated experiment, leaving less room for improvement.

GR/AbstFactory recorded improvements in correctness between the pre-test and post-test for Alt version programs. We believe this to be caused by the fact that both the Pat and Alt versions of this program contained an AbstFactory pattern as noted in Table 1, and that the subjects benefited from the course in understanding this pattern.

6.4. Summary

For Observer, we did not find the negative effect that was observed in the original experiment, while for Visitor we had a very strong negative indication. For Decorator also we found fewer harmful effects, while Composite/AbstFactory turned out about the same.

The total and C++ experience of our subjects was roughly comparable to those of the original experiment, but our subjects had less pattern knowledge to start with. The materials, experimental design and design patterns course were the same. The main differences were a more heterogeneous group of subjects (multiple companies), and the programming environment vs. pen and paper.

Having a detailed log of all compilations enabled better understanding of what the subjects were doing while solving their tasks. In particular, several cases were observed in which a subject worked on one solution, discarded it and did something different, whereas in the final code, no trace of the intermediate solution was present.

6.5. Lessons Learned

Our suggested "lessons learned" are similar to those in the original experiment. It is not always useful to use a pattern if a simpler design will do the job; and if the pattern is a complex one like Visitor, even "proper" use can confuse more than it helps.

However, patterns that are more intuitive, such as Observer or Decorator, will generally do little harm, and code based on them can be readily extended even by developers who do not fully understand them.

A developer needs an awareness, not just of design patterns or whatever other design method is in fashion, but also of good design principles in general. Good breadth in education and experience can make up for lack of knowledge of specific patterns, but the opposite does not follow. Educational institutions should avoid the temptation to concentrate too much on the current design fashion, whatever it might be.

7. Methodological Results

During the course of this experiment we identified a number of issues relating to the conduct of such experiments, which we will summarize here:

7.1. Measurements

The subjects measured the total elapsed time for a task themselves, noting it in a questionnaire. They could also add free-form comments at the completion of every work task. Our logging system also measured the elapsed time, and unobtrusively saved time-stamped copies of every file compiled. After each task, all the subjects completed the questionnaire, grading the difficulty of the task, the helpfulness of any patterns present, and their own use of the patterns.

The combination of both machine- and self-measured elapsed time, together with the comments, enabled better verification of the actual measurement. The fact that there were no significant discrepancies increased our confidence that the times were measured correctly.

The free-form comments and the compilation logs were valuable in both grading the correctness of the solutions, and in later qualitative analysis.

7.2. Technical Setup

Setting up a lab with 45–50 workstations is an expensive and laborious task. We instead decided to use a Terminal Server™ configuration, in which each subject would bring his/her own laptop computer (the vast majority of consultants now use them), connect to our network and then work entirely inside the Terminal Server environment. We used four servers with a total of six CPUs and 3GB of RAM. The servers ran at 20–30% CPU load during the experiment. Logs were made of various performance parameters to verify that no major server problems affected the experiment.

The subjects were assigned seating based on their group membership, such that two subjects sitting next to each other never worked on the same task.

On the morning of the first day, the subjects were assisted by Simula Research Laboratory technical support staff to connect their laptop PCs to our network. This operation took from a few to about 15 min per subject, depending on their Windows

configuration. There were three staff members on hand to handle this task, and this proved barely sufficient.

If the organization owning the laptop had a restrictive security policy, this relatively simple reconfiguration became very difficult or impossible to perform. We had 10 terminals ready to accommodate subjects who for any reason could not use their own computer, and seven of them were actually used. To avoid losing subjects it is essential to have such a backup option ready. Switching in the middle of the experiment should be avoided; in our case, if the laptop did not work satisfactorily on our network within a maximum of 15 min, the subject was given a terminal instead and the laptop was never used.

7.3 Programming Environment

The environment was set up with a pre-installed editor/compiler, Web browser, a viewer for PDF documents, and nothing else. Access to other functions, files, etc., was removed or blocked. In this way, we ensured that the subjects had equal working conditions, and also guarded against mishaps with incompatible header files and other setup-related problems. It should be noted that such problems are not easily overcome in the C++ world, where even "standard" headers are incompatible among major compiler manufacturers.

7.4. Big-bang Experiments

This experiment had a design that forced all the subjects to be present at a certain place, and at the same time, for several days. Finding participants is quite difficult, because people have to be available at exactly the given time, which may be difficult to schedule for their employers. In addition, if anything happens to prevent attendance, the subject is lost: there is no second chance.

An alternative model is to perform the experiment in batches, over an extended period of time. This is more flexible and robust, and also allows the experiment to be extended with more subjects if needed. However, it precludes experimental designs that require the subjects to have some common activity, such as the patterns course in this experiment.

7.5. Place of Experiment

Due to the design of the experiment, our subjects had to come to Simula Research Laboratory, both for the course and to do the exercises. An alternative model is to use a web-based tool (Arisholm et al., 2002b) to deliver the programming tasks, questionnaires and tools directly to the subjects, and have them perform their tasks in their own office environment.

The experimenters must in any case be in attendance on the premises to handle any problems.The major methodological challenge in this model is to keep control of the experiment; on the technical side it is not trivial to make sure that all the subjects' computers have equivalent and properly working environments. It is also more difficult to install and use various monitoring tools.

7.6. Recruitment and Subject Selection

The intended population for this experiment were programmers who make general-purpose data processing software. A number of consultancy companies were asked to participate in the experiment. They were told the general outline, consisting of a one-day patterns course between two half days of exercises. The subjects were paid for the exercise time, and received the course free.

Both the companies and individual subjects were told that they were participating in an experiment, but were not told about any of the goals, hypotheses or expectations, nor what was measured or how.

The participation was formalized by contracts signed by each company and Simula Research Laboratory. Each company was also allowed to charge a limited number of management/overhead hours. Our experience is that allowing reasonable overhead costs increases the likelihood of participation.

The experimental design required all participants to be present at the same time, for two consecutive days. This made it much more difficult to get a sufficient number than for previous experiments that could be done on each company's premises, one company at a time.

7.7. Subject Background Mapping

The background data were collected using an on-line questionnaire prior to the experiment (Arisholm et al., 2002a). The questionnaire used was identical to that used in the original experiment. However, instead of the subjective, manual process that was used to allocate subjects to group in the original experiment, the answers were transferred to a database, and processed by a scoring program. A score was calculated in each of the following three categories:

1. C++ programming experience and volume, measured in number of programming years and number of lines written.

2. Knowledge of design methods, measured by number of methods known and number of practical uses of each method.

3. Knowledge of patterns, measured by number of patterns known, and number of practical applications of each pattern.

Finally, the scores were combined with relative weights of $^1/_6$, $^1/_6$ and $^2/_3$, reflecting the relative priorities of the categories.

Using a scoring program made the relative weights of the different factors explicit and visible. However, no automated process can fully address all details of a subjects' qualifications, especially free-form comments that clarify the purely quantitative aspects. Manual inspection and sometimes adjustment is required, for example, if a subject filled in "5" as the number of years of education, but stated that two of them were in high school.

7.8. Prequalification and Blocking

Having too large a spread of education, experience and knowledge in the subjects can undermine the use of standard quantitative statistics, because it introduces individual differences that may mask the trait we seek to measure. As experienced in this experiment, it is difficult to predict the performance of subjects from indirect measures such as education or work experience, and balancing according to such criteria may therefore not be enough to ensure an actual balance with respect to the experimental tasks.

This can be mitigated by using nontrivial familiarization and calibration tasks that are common to all the subjects. It might also be necessary to exclude some subjects based on a pre-test, if it turns out that they are not qualified to take part. The sample is then no longer random, and great care must be taken to ensure that the subjects are (and remain!) a representative sample of the population that is being studied. We note that simply relying on participating organizations to select a sample (by deciding who to send) may not be enough.

One consequence is that the process of selecting subjects should be started several months in advance of the experiment, to allow time for sufficient iterations. "Big-bang" experiments are especially sensitive, because there is no second chance. As discussed in Section 4.3, this experiment was robust with respect to (lack of) group balancing. However, mapping the subjects' background is still important to help ensure a representative sample of the targeted population, and thereby improve the experiments' external validity.

7.9. Details Matter

In pen and paper experiments, subjects do not have to write exactly correct syntax. Once a compiler is introduced this is no longer true, and technicalities that are viewed by the experimenter as irrelevant may consume significant amounts of time.

In the BO program, several subjects knew neither the truth table nor the C++ syntax for the `XOR` operator, and consequently spent time on this.

In the GR program, one subject had never seen a pen plotter and consequently did not understand what to do. The experimenter therefore has to take extreme care to fully specify, document or avoid such details to improve data correctness.

The experimental design used here seeks to mitigate the consequences of such "details". All the three problems cited contributed to increase the variability of the data, or to data loss. By making multiple measurements on both Pat and Alt programs and tasks for each subject, the overall impact is minimized and the chance of systematic bias in the results lessened.

8. Threats to Validity

An experiment is by definition an artificial situation. In this section, we address threats to both internal and external validity of the observed results. The discussion is in part based on the guidelines recently set forth by Kitchenham et al. (2002).

8.1. Threats to Internal Validity

Internal validity is the degree to which the observed effects depend only on the intended experimental variables. In this experiment, the main threats are from inter-individual, and inter-group, differences between subjects that mask the intended effects. The purely technical proficiency of the subjects (as distinct from their ability to understand program structures and patterns) is also a factor.

8.1.1. Group Balancing

We have already described how the groups were balanced with respect to pattern knowledge, general programming experience and C++ experience. Recent experience with the programming environment (Microsoft Visual Studio 6.0) and Windows itself was checked and also found to be reasonably well distributed.

The regression model and estimation method chosen are robust with regard to group differences, since each subject is its own control: all the subjects perform the work tasks of all the programs (half Alt and half Pat versions).

As it turned out (see 4.3), the regression model revealed no significant effect of the pre-qualification score, so a fully random assignment would probably have been just as good. In an experimental design where each subject receives either treatment A or B, group balancing is much more important. However, balancing can never be relied on completely, and is susceptible to problems such as participants not turning up. In such cases, some kind of calibration task is needed to determine actual performance levels, and should be included in the statistical model (Arisholm et al., 2001).

8.1.2. Technical Factors

The programs used for the tasks were originally designed with a minimal user interface that consisted of a declaration of a `Window` class with a few self-

explanatory methods (`drawtext, drawline, erase, resize`). In the present experiment, that class was implemented so that the programs actually ran and could show a window; the well-known stream mechanism for creating console text output `cout << ''Hello\n''`; was also added.

The windows so constructed remained active even if the program was stopped in a debugger, so their output was visible. To avoid distractions, the implementation was carefully hidden. No knowledge of Windows, Microsoft Foundation Classes, Java or any other specific system was assumed or needed.

Logs were kept of server load parameters to determine if overloads or faults interfered somewhat with the experiment. Moreover, all compilations were inspected, and subjects were also encouraged to submit (as free text) comments about any technical obstacles they might have encountered.

8.1.3. C++ Proficiency

All the subjects performed an initial, familiarization task in order to try out the programming environment and the user interface.

Individual differences in C++ experience and capabilities interfered with the time spent on the programming. C++ is a relatively complex language with fine syntactical and semantic distinctions. Developers who are not proficient can spend significant time on details.

To lessen this threat, all compilations were evaluated, and the editing time for compilations that were purely about syntactical problems was subtracted from the total time for each subtask. The procedure is described in Section 4.1 above. Comparisons of estimates of the regression model coefficients and their confidence intervals showed that the corrections added no systematic bias.

However, neither did they shrink the confidence intervals much. Corrections of this kind will always depend to some extent on the subjective judgment of those doing the grading. Also, since copies of the source files were made only when compilations were performed, any subjects who spent time on syntax without making compilations cannot be assigned corrections. For these reasons the corrections were not used in the final analysis.

Another factor, likewise determined by inspection of the solutions, is that several subjects implemented one solution to a large extent, only to abandon it and start again in a different way. This also increased the individual variations.

8.2. Threats to External Validity

External validity is the degree to which the results can be generalized and transferred to other situations. Several differences between the experimental situation and real-world maintenance must be considered.

8.2.1. Maintenance by the Original Designers

In our experiment, the maintainers were different from the original programrs, so the experiment is not applicable to maintenance by the original designers. Original designers would be expected to remember not only the actual design, but also much of the motivation for it.

The use of design patterns may not have much impact in that case; in the current context, we are not interested in improvements resulting simply from the fact that a design with patterns fits the problem better than some alternative design.

8.2.2. Design Pattern Experience

Some maintainers may have more experience with design patterns than did our subjects. In that case, we would expect the beneficial effects of patterns to be greater. Thus, the experiment is conservative in estimating benefits of using design patterns.

This expectation is motivated by the significant improvements (with one exception) in maintenance speed after the patterns course. The exceptions can largely be explained by the course being too short, so that subjects attempted to use a pattern that they did not fully understand. Deeper knowledge will probably not make the situation worse!

8.2.3. Program Size, Task Size and Tools

Real-world programs are much larger than those used in the experiment. With a restricted time and money budget, this is a limitation that is difficult to overcome. In addition, real-world programs are sometimes less well documented, and changes may be larger and involve more than one pattern. The effects of such differences are difficult to predict on a general basis. There are undoubtedly interaction effects that can occur between patterns, but would not be visible in such an experiment.

Judging by the qualitative findings, we would expect the results to be generally transferable. Good or bad understanding of a structural pattern implies similarly good or bad understanding of the structure of programs employing it, and any benefits or problems should therefore be applicable. As large changes are often made up of several small ones, program size will be a scaling factor, but will probably not alter the direction of the effect. Relatively small change tasks, of the same order of magnitude as those in the experiment, do actually occur in industrial settings (Arisholm, 2001).

Maintainers may be more familiar with the language, development tools, etc., than were some of our subjects. However, given the widespread use of consultants, and relatively high turnover of developers in general, the experiment may reflect the real world in this respect more accurately than one might desire.

8.2.4. *Realism of Tasks*

In the experiment, most work tasks consisted of adding features that corresponded in some way to features or functions already in the code. The subjects could therefore use parts of the existing code as templates for their solutions. In an industrial context this would not necessarily be true.

However, most software does not fundamentally change its nature during what is normally termed "maintenance". Most features added or modified during maintenance correspond to something already present; for instance, a new layout for an existing screen, the addition of a field, or the addition of another web page to an existing structure. We therefore do not consider this aspect of the experiment to be a significant threat, though it excludes "maintenance" such as porting a program to a new platform or rewriting it for a fundamentally different kind of database.

8.2.5. *Domain Knowledge*

Each program came from a different domain: Graphics, Formula manipulation, GUI/Presentation and Communication. The design patterns used (AbstFactory, Composite, Visitor, Observer and Decorator) are not domain-specific. Lack of domain knowledge was therefore more a threat to internal than external validity.

Inspection of the code logs and comments made by the subjects revealed a few cases of domain unfamiliarity: one subject did not know what a pen plotter was, and several subjects had some problems with Boolean arithmetic (see Section 7.9). While this increased the uncertainty of the estimates of the relevant coefficients of the regression model, we do not consider it a threat to the external validity of the results concerning the patterns themselves.

8.2.6. *Experimental Stress*

Finally, our subjects were working under artificial, experimental conditions. The closest analogy is an exam. Even if one were always sitting beside someone working on a totally different task, watching a relaxed neighbor while one struggles is quite stressful, on top of the knowledge that one is being measured in visible, and possibly invisible, ways.

Another difference is that it is possible to walk away from a nonworking solution, as actually happened with several subjects. In industry that option is simply unavailable. When it happens, projects tend to become highly visible failures.

Working under stressful and tight deadlines is not unusual in industry (Yourdon, 1999). The additional stress from the experimental situation is expected to add to the size of the individual differences, because some individuals handle the situation better than others. Given the design of the experiment, we do not expect the results to be systematically skewed by this, so they should still be transferable to an industrial context.

9. Conclusions

We replicated the experiment performed by Prechelt et al. (2001), which investigated the question whether it is useful (with respect to maintenance) to design programs using design patterns, even if the actual design problem is simpler than that solved by the pattern. Our replication sought to increase experimental realism by using a real programming environment instead of pen and paper, and by using paid professionals from multiple consultancy companies as subjects.

Logging tools were used to collect copies of the evolving solutions while the subjects worked. Together with free-form comments made by the subjects, this formed the basis for a qualitative evaluation of the results. In addition, a regression-based approach using GEE was adopted for the quantitative statistics. This approach takes into account the correlations between multiple work tasks performed by each subject.

We found that each design pattern tested has its own nature, so that it is not valid to characterize design patterns as useful or harmful in general, at least in the context (maintenance by other programmers than the original developers) addressed here.

The Observer and Decorator patterns were generally understood even by subjects with little or no previous patterns knowledge, and after a short course the value of the patterns, in terms of both development time and, to some extent, correctness of the solutions, increased. The Composite pattern, with its reliance on recursion, caused some problems. It may be that recursion is no longer in general use in this kind of software, and a possible cause is the availability of predefined container classes in most languages. The Visitor pattern, which has a fairly complicated structure, extracted a high cost in development time and poor correctness. Many subjects actually ignored it even when presented with template solutions that used it (and were documented as such).

Our results differ somewhat from those found by Prechelt et al. (2001), especially in the case of Visitor and Observer. While they found Visitor to be without significant harmful effects, few of our subjects achieved a good solution with it, even after the course. By contrast, we observed no significant harm done by using the Observer pattern.

Having not only the final solution, but also the intermediate steps (provided by the logging mechanism), made possible a more extensive quantitative analysis. Using a real programming environment was one prerequisite for such logging. The realism was also increased by introducing the need to compile and test the solutions.

We also demonstrated that it is possible to perform experiments on this scale while using a realistic environment and tools, and professional, paid subjects. It is possible to use Windows Terminal Server to provide a pre-configured environment without having to set up each workstation individually. This can be combined with Web-based tools to deliver content and questionnaires to subjects, thereby enabling experiments with a larger number of subjects.

Paying subjects to participate, and allowing some overhead costs, make it possible to get professional developers as subjects. If the experimental design allows it, it is also possible for them to participate while being in their normal work environment.

In future work, these factors can be combined to increase the realism of the experiments and address some of the traditional threats to external validity.

Only four design patterns were evaluated in the original experiment and this replication. One area for future work is to evaluate other design patterns in widespread use from a similar standpoint: what effect would their use have on future maintenance, for programrs with and without relevant design pattern knowledge. Another need is to evaluate design patterns in larger contexts. The programming tasks do not necessarily have to be much larger, but the software of which they are a part should be of a more realistic size.

Acknowledgments

Gunnar J. Carelius, Halvard Moe and Åsmund Ødegård of the Simula Research Laboratory technical staff provided absolutely invaluable assistance on the technical side. Under intense time pressure they courteously assisted experiment participants in setting up their laptops to run on the Simula network. Gunnar also set up the Terminal Servers and provided other technical assistance.

Lene Hansen smoothly handled the practical logistics of travel, lunch and other practical details.

Our participants approached the experiment with great enthusiasm and worked intensely to complete their work tasks.

Notes

1. All data are stored in a relational database together with the relevant source files, so that it is possible at any time to retrieve data with or without any corrections and grades, inspect the classification of each compilation, and the file difference giving rise to it.
2. Notation: The hypothesis is identified by its letter/number combination as in Tables 3, 4, 6 and 8 followed by the direction of the expected effect, and observed effect in %. The lower and upper limits of the 95% confidence interval are given as subscript and superscript.
3. The subtle interaction effects which it is possible to encounter with multiple observers and event-driven programming in general, become visible only in programs of greater complexity.

References

Alexander, C. 1978. *A Pattern Language: Towns, Buildings, Construction*. New York, USA: Oxford University Press, Inc.

Alexander, C. 1987. *The Timeless Way of Building*. New York, USA: Oxford University Press Inc.

Arisholm, E. 2001. Empirical assessment of changeability in object-oriented software. Ph.D. thesis, University of Oslo.

Arisholm, E., Sjøberg, D. I. K., Carelius, G. J., and Lindsjørn, Y. 2002a. SESE an experiment support environment for evaluating software engineering technologies. In *NW-PER2002 (Tenth Nordic Workshop on Programming and Software Development Tools and Techniques)*. Copenhagen, Denmark, pp. 81–98.

Arisholm, E., Sjøberg, D. I. K., Carelius, G. J., and Lindsjørn, Y. 2002b. A web-based support environment for software engineering experiments. *Nordic Journal of Computing* 9(4): 231–247.

Arisholm, E., Sjøberg, D. I. K., and Jørgensen, M. 2001. Assessing the changeability of two object-oriented design alternatives—a controlled experiment. *Empirical Software Engineering* 6, 231–277.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture*. Chichester: Wiley.

Diggle, P., Liang, K., and Zeger, S. 1994. *The Analysis of Longitudinal Data*. Oxford: Oxford University Press.

Efron, B., and Tibshirani, R. J. 1993. *An Introduction to the Bootstrap*. Monographs on Statistics and Applied Probability. London: Chapman & Hall.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El-Emam, K., and Rosenberg, J. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28(8): 721–734.

Liang, K., and Zeger, S. 1986. Longitudinal Data Analysis using Generalized Linear Models. *Biometrika* 73, 13–22.

Lindsay, R., and Ehrenberg, A. 1993. The design of replicated studies. *The American Statistician* 47(3): 217–228.

McCullagh, P., and Nelder, J. 1989. *Generalized Linear Models*. New York: Chapman and Hall.

Prechelt, L. 2000. An empirical study of working speed differences between software engineers for various kinds of task. Submitted to *IEEE Transactions on Software Engineering*, to be revised.

Prechelt, L., Unger, B., Tichy, W. F., Brössler, P., and Votta, L. G. 2001. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering* 27(12): 1134–1144.

Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E., and Vokác, M. 2002. Conducting realistic experiments in software engineering. In *ISESE2002* (*First International Symposium on Empirical Software Engineering*). Nara, Japan, pp. 17–26, IEEE Computer Society.

Smith, D., Robertson, W., and Diggle, P. 1996. Object-oriented software for the analysis of longitudinal data in *S*. Technical Report Technical Report MA96/192, Department of Mathematics and Statistics, University of Lancaster.

Yourdon, E. 1999. *Death March*. Indianapolis, Indiana: Prentice Hall PTR.

Marek Vokáč completed his M.Sc. degree in computer science at the University of Oslo in 1992, and is currently working on his Ph.D. in the same field, at the Simula Research Laboratory in Oslo. He has been working as a professional software developer since 1985, on both minicomputers, PC's and client/server systems. His interests center on software engineering, software design, databases and design patterns.

Walter F. Tichy has been professor of Computer Science at the University Karlsruhe, Germany, since 1986. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and served six years on the faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is currently directing research on a variety of topics, including empirical software engineering, autonomic computing, software configuration management, cluster computing, optimizing compilers for parallel computers, and optoelectronic interconnects. He has consulted widely for industry.

He earned an M.Sc. and a Ph.D. in Computer Science from Carnegie Mellon University in 1976 and 1980, resp. He is director of the Forschungszentrum Informatik, a technology transfer institute. He is co-founder of ParTec AG, a company specializing in software for computer clusters. He was program co-chair for the 25th International Conference on Software Engineering (2003). Dr. Tichy is a member of ACM, GI, and the IEEE Computer Society.

Dag Sjøberg received an M.Sc. degree in Computer Science from University of Oslo in 1987 and a Ph.D. degree in computing science from University of Glasgow in 1993. He has five years industry experience as consultant and Group Leader. He is now Research Manager of the Department of Software Engineering, Simula Research Laboratory and Professor in Software Engineering in the Department of Informatics, 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.

Erik Arisholm received a M.Sc. degree in Electrical Engineering from University of Toronto and a Ph.D. degree in Computer Science from University of Oslo. He has seven years industry experience in Canada and Norway as a Lead Engineer and Design Manager. He is now Research Group Leader for Object-Oriented Analysis and Design in the Department of Software Engineering, Simula Research Laboratory and Associate Professor in the Department of Informatics, 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 controled experiments in software engineering.

Magne Aldrin is a chief research scientist in the Department of Statistical Analysis, Pattern Recognition and Image Analysis at the Norwegian Computing Center, Oslo, Norway. His research interests are within regression analysis and general statistical methodology. He obtained his Ph.D. from the University of Oslo, Norway.