# A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions

Lutz Prechelt, *Member*, *IEEE Computer Society*, Barbara Unger, *Member*, *IEEE*, Walter F. Tichy, *Member*, *IEEE*, Peter Brössler, and Lawrence G. Votta, *Member*, *IEEE*

**Abstract**—Software design patterns package proven solutions to recurring design problems in a form that simplifies reuse. We are seeking empirical evidence whether using design patterns is beneficial. In particular, one may prefer using a design pattern even if the actual design problem is simpler than that solved by the pattern, i.e., if not all of the functionality offered by the pattern is actually required. Our experiment investigates software maintenance scenarios that employ various design patterns and compares designs with patterns to simpler alternatives. The subjects were professional software engineers. In most of our nine maintenance tasks, we found positive effects from using a design pattern: Either its inherent additional flexibility was achieved without requiring more maintenance time or maintenance time was reduced compared to the simpler alternative. In a few cases, we found negative effects: The alternative solution was less error-prone or required less maintenance time. Although most of these effects were expected, a few were surprising: A negative effect occurs although a certain application of the Observer pattern appears to be well justified and a positive effect occurs despite superfluous flexibility (and, hence, complexity) introduced by a certain application of the Decorator pattern. Overall, we conclude that, unless there is a clear reason to prefer the simpler solution, it is probably wise to choose the flexibility provided by the design pattern because unexpected new requirements often appear. We identify several questions for future empirical research.

**Index Terms**—Controlled experiment, design pattern, design alternatives, maintenance, change effort.

## 1 Introduction

Object-oriented design patterns as presented by Gamma et al. [7] are becoming increasingly popular. Their purpose is capturing design knowledge in such a form that it can be reused easily, even by less experienced designers.

Most design patterns collected in the popular book by Gamma et al. [7] aim at reducing coupling and increasing flexibility within systems. For instance, many of the patterns delay decisions until runtime that would otherwise be made at compile time or they factor functionality into separate classes. As a consequence, they often allow adding new functionality without changing old code.

Besides offering proven solutions using patterns purportedly provides additional advantages: Design patterns define terminology that improves communication among designers [1] or from designers to maintainers [7]. They also make it easier to think clearly about a design and encourage the use of "best practices."

Our work aims at testing and evaluating these claims.

- *L. Prechelt, B. Unger, and W.F. Tichy are with the Fakultät für Informatik, Universität Karlsruhe, 76128 Karlsruhe, Germany. E-mail: {prechelt, unger, tichy}@ira.uka.de.*
- *P. Brössler is with sd&m GmbH and Co., KG Thomas-Dehler-Str. 27 81737 München, Germany. E-mail: Peter.Broessler@sdm.de.*
- *L.G. Votta is with Motorola Inc., 1501 W. Shure Dr., MS SW Atrium, Arlington Heights, IL 60004. E-mail: votta@cig.mot.com.*

*Manuscript received 11 Jan. 1999; revised 12 May 2000; accepted 25 Oct. 2000.*
*Recommended for acceptance by M. Jazayeri.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 108962.*

### 1.1 Isn't This Just Obvious?

Many readers may question the need for an empirical study of a technique whose mechanism and benefits are so obvious: "Clearly patterns do have the advantages claimed for them!" However, as software engineers have discovered before, sometimes spectacularly (e.g., in the case of multi-version programming [8]), our intuition may seriously mislead us and words such as "clearly" and "obviously" do not constitute confirmation.

Hence, as scientists, we should seek solid and scrutinable empirical evidence instead of relying on anecdotes from irreproducible situations. In mature scientific disciplines, this is a standard procedure before any theory will be considered valid. The present work provides such empirical evidence, indicating that the use of certain design patterns can indeed, as expected, improve the maintainability of programs.

However, our work also produced two results that appear nonobvious (at least upon first look) and, hence, produce useful insights. First, it provides a specific example where reasonable use of a design pattern made a program harder to maintain. This case can serve as a starting point for empirically grounded development of guidelines for the use of patterns. Second, we observed that, compared to a straightforward solution to a problem, a design that provides *unnecessary* flexibility may still be easier to maintain. We consider both results to be not "just obvious."

### 1.2 The Complexity Trade-Off

The following thought leads to our experiment approach. Given the popularity of the Gamma et al. design patterns,

one can expect that they will often be used in situations where their flexibility is not needed: The pattern solves the problem but is more powerful than required.

In such situations, there are two competing forces: On the one hand, applying the pattern might be a good idea because of the advantages of common terminology, proven solutions, and best practices. On the other hand, it may be a bad idea because the solution applied may be more complicated than necessary and may thus make understanding and change more difficult, in particular for programmers who have not learned about design patterns before. The experiment described here investigates this trade-off.

### 1.3 Experiment Overview

Our controlled experiment assesses designs using patterns versus alternative designs in the context of program maintenance. We consider four different programs with different design patterns. Among the flexibility and functionality properties of the design pattern solution of each program, at least one is not actually needed for the given maintenance tasks. For each program, the experiment compares the performance of two groups of subjects on these maintenance tasks. Two different baseline program versions are used: Version PAT applies design patterns, whereas version ALT employs a simpler solution that exhibits only the functionality and flexibility actually required.

We use well-documented, modestly-sized, artificial programs that contain implementations of the design patterns ABSTRACT FACTORY, COMPOSITE, DECORATOR, FACADE, OBSERVER, and VISITOR as described in the book [7]. The subjects are professional software engineers. We compare different groups of subjects before and after a two-day design pattern course.

### 1.4 Related Work

A great deal of work is currently being done in both scientific and industrial context towards identifying design patterns, writing them up, discussing and teaching them, building support tools, etc. [1], [2], [3], [6]. Reports on the effects of patterns are available in anecdotal form from various practitioners [1], but there has been little work done in a quantitative fashion, let alone in a controlled environment.

In fact, the only quantitative, controlled experiment on patterns reported so far appears to be [10] (see [9], [11] for details). It investigates communication improvements through patterns in a maintenance situation. Maintenance can be done quicker and with fewer errors if design patterns are explicitly documented. This result confirms some of the purported positive effects on communication, but does not address effects of patterns on actual software structure.

### 1.5 Article Overview

In Section 2, we describe the experimental design, the underlying performance model, the subjects' background, and how the experiment was done. We also discuss the internal and external validity of the experiment. Section 3 describes the programs used in the experiment, the work tasks, and the expected and actual results. We will assume that the reader understands the relevant design patterns and their properties; thus, we will not describe them. The conclusion sketches the common denominator of the results, possible consequences for proper program design, and further research questions to be investigated.

## 2 Description of the Experiment

We now give a short description of the experiment design and conduct. More detail, including the original experiment documents such as the programs, the work tasks, and the raw result data are available from http://wwwipd.ira.uka.de/EIR/.

### 2.1 Experiment Objectives

It is tempting to use design pattern solutions even if the actual design problem is simpler than the one solved by the pattern. In this experiment, we consider the case that not all of the flexibility of a particular design pattern is needed in a program. Therefore, the solution based on patterns could be replaced by a simpler one. We want to test whether still using the design pattern in such cases is helpful or harmful. We compare design pattern solutions versus alternative solutions for programs involving different design patterns and for subjects having different levels of pattern knowledge.

### 2.2 Hypotheses

Our hypotheses that will be spelled out informally as *expectations* take the form that a design pattern P does not improve the performance of subjects doing a maintenance exercise X on program A (containing P) when compared to subjects doing the same exercise X on an alternative program A' (not containing P). The "helpful," "harmful," or "neutral" interpretations are derived from the rejection or nonrejection of these hypotheses.

### 2.3 Design

Our experimental design uses three independent and two dependent variables. The independent variables are the programs and change tasks, the program version, and the amount of pattern knowledge; the dependent variables are time and correctness.

- **"Program and change task."** We use four different programs, each with a different purpose, different design patterns, and two or three different maintenance tasks.
- **"Program version."** There are two different, functionally equivalent versions of each program. One version (named "pattern version," PAT) employs one or more design patterns, the other (named "alternative version," ALT) represents a somehow simpler design using fewer design patterns or simplified versions of them. This is the central variable of our experiment. However, the subjects did not know that this variable was in the experiment at all; they only knew "the experiment tests the usefulness of patterns."

TABLE 1
Order of Programs Per Group

| temporal sequence | | group A | group B | group C | group D |
|---|---|---|---|---|---|
| PRE | 1st problem | ST PAT | GR PAT | CO ALT | BO ALT |
| PRE | 2nd problem | GR ALT | ST ALT | BO PAT | CO PAT |
| pattern course | | | | | |
| POST | 3rd problem | CO ALT | BO ALT | ST PAT | GR PAT |
| POST | 4th problem | BO PAT | CO PAT | GR ALT | ST ALT |

*ST, BO, CO, and GR are the programs and ALT or PAT indicates which program version was used. See descriptions in the text. For instance, in the context of program BO, we will call group D the PRE-ALT group, group B the POST-ALT group, B+D together the ALT group, C+D together the PRE group, etc.*

- **"Amount of pattern knowledge."** This is the difference between pretest and posttest. The experiment had two parts on two different days. The first part (the pretest, PRE) was performed in the morning of the first day. Then a pattern course was taught during the afternoon and the next morning. In the afternoon of Day 2 the second part of the experiment (the posttest, POST) was performed.

  Before the experiment, the participants had only little pattern experience; about half of the participants had no pattern knowledge at all. Therefore, the posttest represents subjects with significantly higher pattern knowledge than the pretest.
- **Dependent variable "time."** The time (in minutes) taken for each maintenance task.
- **Dependent variable "correctness."** We decided whether the participant's solutions fulfilled the requirements of the task or not. For many tasks, all groups achieved near-perfect correctness, so we will often ignore this variable.

We divided the subjects into four groups. In both pretest and posttest, each group maintained one PAT program and one ALT program, with two or three work tasks for each. Overall, each subject worked on all four programs, and each program was used as often in the pretest as in the posttest and as often in the PAT version as in the ALT version. The design is summarized in Table 1.

## 2.4 Performance Model

For this experiment, we consider the time required for a task to be the sum of the following components:

1. understanding the task,
2. finding out which parts and aspects of the program are relevant to the task,
3. understanding these relevant aspects of the program,
4. understanding how to perform the requested change (change design), and
5. performing the requested change (change implementation).

The first of these components is identical for the PAT and ALT version, the others may depend on the actual program structure. For the second and later tasks to be performed on one program, parts of the time components 2 and 3 may be reused. The five components may contribute rather differently to the overall task completion time, depending on the particular program and task.

## 2.5 Subjects and Groups

The 29 subjects are all professional software engineers. On average, they had worked as software professionals for 4.1 years and their average C++ experience was 2.4 years. Their work typically had a good mix of design, coding, test, and maintenance activities. Fifteen subjects already had some pattern knowledge before the course.

Data about programming and working experience was gathered by a questionnaire weeks before the course. Based on the questionnaire's results, the prospective 32 subjects were carefully assigned into four groups so as to balance as well as possible the professional experience, C++ experience, and, in particular, the level of knowledge of the relevant patterns. Four registered subjects did not appear at the experiment. One additional participant appeared on short notice and was assigned ad-hoc.

The resulting actual group sizes were six to eight subjects in each group, with two to three having theoretical or practical pattern knowledge of the relevant patterns before the course. For more detailed information about the groups, see Fig. 1 and Table 2.

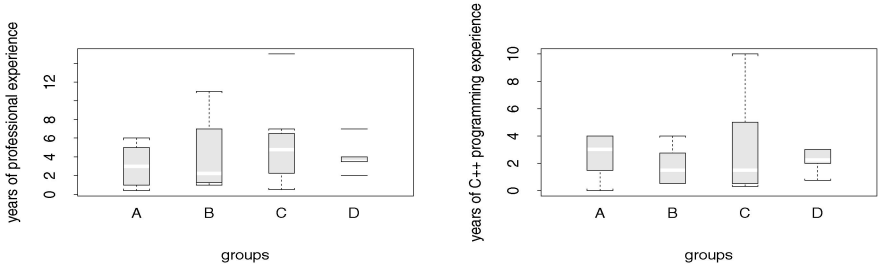


Fig. 1. Left: Years of professional experience of the subjects in each group. Right: Years of C++ programming experience of the subjects in each group.

TABLE 2
Pattern Knowledge of the Groups Before the Pattern Course

| | Abstract Factory | Observer | Decorator | Composite | Visitor |
|---|---|---|---|---|---|
| Group A | $T_{A7}$ | $T_{A5}$ | – | – | $T_{A5}$ |
| Group B | $5_{B1}$ $3_{B3}$ | $5_{B1}$ $T_{B3}$ | $5_{B1}$ | $3_{B1}$ | $T_{B1}$ $T_{B3}$ |
| Group C | $3_{C2}$ $T_{C5}$ | $2_{C2}$ $T_{C5}$ | $T_{C5}$ | $1_{C2}$ $T_{C5}$ $T_{C7}$ | $T_{C5}$ |
| Group D | – | $T_{D3}$ $T_{D4}$ | $T_{D4}$ | $T_{D3}$ $3_{D5}$ | $1_{D5}$ |

*Each subject with pattern knowledge of the relevant patterns before the course is listed with the subject label given as the subscript to the number of times s/he has previously used the pattern. T means that the subject has only theoretical knowledge of the pattern.*

### 2.6 Experiment Conduct

The experiment was performed in November 1997 with sd&m personnel in Munich. The pretest started at 9:30 am, the posttest at 12:40 pm the next day. The subjects worked asynchronously and there was no time limit for completing the tasks; all subjects finished their tasks within three hours.

The subjects received all documents printed on paper: general instructions, a program description, a program listing, work task descriptions, and a postmortem questionnaire. The solutions were delivered in handwriting. The overall amount of text written was small so handwriting speed was not a limiting factor.

### 2.7 Threats to Internal Validity

Internal validity is the degree to which the observed effects depend only on the intended experimental variables. Due to the small group sizes, we must be concerned whether groups differed significantly. Relevant aspects of similarity are overall software capabilities, C++ capabilities, and previous pattern experience. We have reduced differences by balancing the groups explicitly using random blocked assignment (often somewhat misleadingly called randomized blocking although not the blocking is random, rather the assignment of each block's members to the groups is [4]). The blocking was performed in a subjective manual process based on the substitute measures of capability available from the preexperiment questionnaire, as described in Section 2.5. Despite later subject loss (right at the start of the experiment), the resulting groups appear reasonably similar and our results give no reason to believe the opposite. Furthermore, an analysis of variance (see Section 3) indicates that only a small fraction ($\approx$ 7.5 percent) of variation is explained by interpersonal differences of subjects, anyway.

A second consideration is the precision and accuracy of the time stamps recorded by the subjects. By cross-checking, we found these data to be highly accurate and reliable.

### 2.8 Threats to External Validity

External validity is the degree to which the results are generalizable, i.e., transfer to other situations. Several differences to real software maintenance situations limit the generalizability of this experiment: First, the original designers and implementors may be the ones who maintain the program. This was not the case in our experiment and our results do not apply to such cases. The maintainers may also have more pattern experience than our participants. The consequences of this difference are unclear; but we do not believe them to be dramatic. Second, real programs will often be less well documented than the experiment programs, real programs are typically larger, and change tasks rarely revolve closely around a design pattern. The effects of such differences probably differ from one case to the next. Third, real maintainers implement and test their solutions (instead of only writing them on paper), that will typically trade some of the incorrectness observed in the experiment against additional time. Furthermore, without an explicit theory of SW maintenance, it is difficult to predict what effect other design patterns (and alternatives) than the five specific ones used in the experiment may have.

## 3 Results

First, we perform an analysis of variance (ANOVA) for identifying the variables that are most relevant for explaining work time. Table 3 shows the most relevant factors found. Clearly, the differences between the various work tasks are most important. The next most important variables are the difference between PAT and ALT for each

TABLE 3
Analysis of Variance (AOV) of Work Time

| | DF | Sum Sq | Mean Sq | F Value | Pr(F) |
|---|---|---|---|---|---|
| worktask | 8 | 32570 | 4071.2 | 59.153 | <0.001 |
| worktask*PatAlt | 7 | 3700 | 528.5 | 7.679 | <0.001 |
| worktask*PrePost | 8 | 2079 | 259.8 | 3.775 | <0.001 |
| worktask*correctness | 13 | 2861 | 220.1 | 3.197 | <0.001 |
| subject ID | 28 | 4444 | 158.7 | 2.306 | <0.001 |
| order | 1 | 40 | 40.2 | 0.584 | 0.446 |
| residuals | 195 | 13421 | 68.8 | | |

*The variables for the model are the work task (per program), PAT or ALT, PRE or POST, correctness of the solutions (on a 4-point ordinal scale), subject ID, and order (first or second test of session). Order is not significant and the contribution of subject identity is relatively small; the other factors deserve separate discussion.*

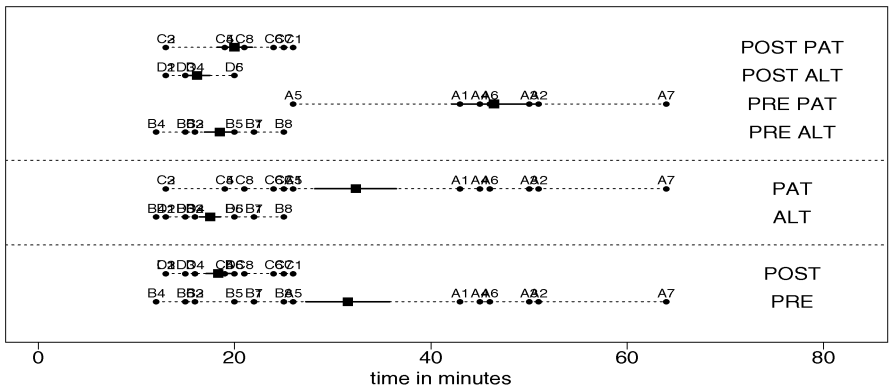


Fig. 2. Time in minutes required for program *Stock Ticker* work task 1. Each dot marks one subject, the square is the arithmetic mean, the line indicates plus/minus one standard error of the mean. The top part (four lines) shows the four individual groups. The middle part (two lines) shows the same but with the PRE-PAT and POST-PAT groups combined and the PRE-ALT and POST-ALT groups combined. Likewise, the bottom part (two lines) shows PAT plus ALT groups combined.

task (with a mean contribution about one-eighth as large), and the difference between PRE and POST for each task (one-sixteenth as large). Therefore, our discussion will be structured along these variables. As we will see below, the correctness differences are relevant for few worktasks only. The systematic interpersonal differences (as described by the influence of the variable subject ID) are not dramatically large. The order difference between the first and second program within the pretest or within the posttest is not a significant factor at all.

Hence, for the further analysis, the results are discussed worktask by worktask. For each of the programs, we describe the program and its use of design patterns, the work tasks and solutions, and the expected and actual results.

The expectations form the basis of our discussion and interpretation of the quantitative results. Note that this style of discussion is dense, but no simpler form would adequately characterize the effects we observed.

For the actual statistical analysis, we did not want to rely on the assumption of normal distributions as made in the standard analysis of variance techniques. The usually nonparametric techniques, such as Kruskal-Wallis or Wilcoxon test, on the other hand, perform an analysis with respect to the medians of samples, rather than the means. However, we would be more interested in the means because the mean is more relevant for final software development cost. Therefore, for further analysis, we use distribution-free Bootstrap methods [5]. Given two samples $A$ and $B$ of work time values to be compared, we compute a bootstrap distribution[1] of differences of mean work times and directly read p-values from this distribution (percentile method).

## 3.1 Observer: Stock Ticker (ST)

**Program description**. *Stock Ticker* is a program for directing a continuous stream of stock trades (title, volume, unit price) from a stock market to one or more displays that are also part of the program. The displays advertize the information or part of the information.

Both versions of *Stock Ticker* consist of seven classes. The PAT version contains an OBSERVER in which four of the seven classes participate. This version of the program has 343 lines (including comments and blank lines). The ALT version of the program includes one class that contains an instance variable for each display and updates the displays when the data changes. No dynamic registration of observers is implemented. This version has 279 lines.

### 3.1.1 Work Task 1

*"In the given program listing only one of the two concrete display types is used. Enhance the program such that a second display* [of the yet unused display type] *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 a new display instance variable and invoke the displaying of new data on each data update. The main work in this task is to comprehend the structure of the program, in particular how the displays receive data.

**Expectations**. The structure of the PAT version is more complicated than the structure of the ALT program version. When subjects lack knowledge of the OBSERVER pattern (in the pretest) they have to find out how the OBSERVER mechanism works, thus PRE-PAT subjects should require more time than PRE-ALT subjects (we call this expectation "E1"). Given sufficient pattern knowledge, on the other hand, the PAT group may understand the program structure more quickly than the ALT group (E2).

**Results**. Fig. 2 supports E1: PRE-PAT subjects require more than twice as much time than PRE-ALT subjects (151 percent more time, 46.6 minutes versus 18.5 minutes, significance $p < 0.001$). In the posttest, however, the PAT subjects still required more time than the ALT subjects (23 percent more time, 20 minutes versus 16.2 minutes, significance $p = 0.023$), refuting E2. We conclude that, for this application and this type of maintenance tasks, the use of the OBSERVER pattern may be harmful.

1. The bootstrap distribution is computed empirically by doing the following 10,000 times: Given $A$, compute a bootstrap resample $A'$ by taking a random sample of size $|A|$ with replacement. Likewise, compute $B'$ from $B$. The difference of the resample means $(\overline{A'} - \overline{B'})$ becomes one element of the bootstrap distribution.

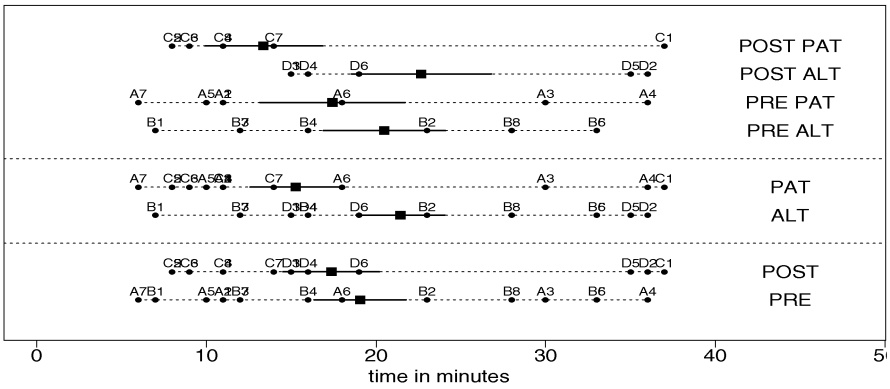


Fig. 3. Time required for program *stock ticker* work task 2.

#### 3.1.2 Work Task 2

*"Change the program so that new displays can be added dynamically at runtime."* The PAT groups only had to realize that nothing needed to be done. The ALT groups had to add the functionality of an OBSERVER (at least two lines had to be changed, one line had to be deleted, and one method had to be added.)

**Expectations**. In contrast to all other tasks in the experiment, this task is clearly unfair; we expect the ALT version to be clearly at a disadvantage (E3). In the PAT version of the program the subjects need to know that the OBSERVER already implements the functionality required. POST subjects should immediately recognize this; PRE subjects might lack the relevant knowledge, slowing them down. The PRE-ALT subjects may have to reinvent the OBSERVER solution and all ALT subjects have to implement it, hence they should be far slower.

**Results**. Fig. 3 confirms E3. The unfair task is completed on average 29 percent faster on the PAT version (15.3 minutes versus 21.4 minutes, significance of difference $p = 0.045$). For both versions, the difference between PRE and POST is not significant.

### 3.2 Composite and Visitor: Boolean Formulas (BO)

**Program description**. *Boolean Formulas* contain a library for representing Boolean formulas (AND, OR, XOR, NOT, and variables) and for printing the formulas in two different styles. Furthermore, it contains a small main program that generates a formula and invokes both printing routines.

The PAT version of *Boolean Formulas* consists of 11 classes spanning 470 lines. The boolean formulas are represented by a COMPOSITE. The printing routines are implemented as VISITORs. For each concrete class of the COMPOSITE, a printing method is implemented in each of the two VISITORs. Each class of the COMPOSITE provides a dispatch method for the VISITORs.

The ALT version of the program is shorter: eight classes spanning 374 lines. It has almost the same structure as the PAT version except for the VISITOR pattern, that is completely missing. The different printing routines are located directly in each COMPOSITE class instead. The VISITOR solution allows for adding new visitors without changing the COMPOSITE classes.

#### 3.2.1 Work Task 1

*"Enhance the program to evaluate the Boolean formulas, i.e., to determine the result for a given formula represented by a COMPOSITE and values of the variables."* The printing routines serve as structural examples. The PAT groups had to create a new VISITOR and the ALT groups had to add new methods to each concrete class of the COMPOSITE.

**Expectations**. In principle, it should be easier to create a single new class similar to another rather than adding a method to several classes. This should favor the PAT groups. However, the VISITOR pattern is technically quite difficult to understand. We expect that it will take more time for the PAT groups to understand the current application of the VISITOR pattern than for the ALT groups to find where to add the methods (E1). Gaining pattern knowledge should help all groups (E2) because even in the ALT program a COMPOSITE is present, so the POST subjects presumably understand the structure of the formula representation faster. The PAT group might profit more from the pattern course than the ALT group (E3) because the working mechanism of the VISITOR is confusing.

**Results**. As one can see from Fig. 4, POST-ALT group is 30 percent faster than the POST-PAT group as expected (29.5 minutes versus 42.4 minutes, significance $p = 0.034$), confirming a part of E1. However, in the pretest, there is a trend in the opposite direction (11 percent slower, 52.2 minutes versus 47.1 minutes, albeit no significant difference, $p = 0.299$), rejecting the other part. Probably in PRE-PAT, the VISITOR is largely just taken for granted and imitated by the subjects (instead of analyzed and understood) and, thus, does not increase complexity. This also explains why the PAT group does *not* profit more from the pattern course than the ALT group (thus, E3 is wrong), although both show some improvement as expected in E2. Overall, it may be that an unrequired VISITOR, although it appears complicated, is not necessarily harmful—but, the data is not quite conclusive in this respect.

#### 3.2.2 Work Task 2

For the second task of this program our instructions were insufficiently clear. As a result, most subjects completely misunderstood the job and delivered something very different from what we had intended. We therefore ignore the task here.

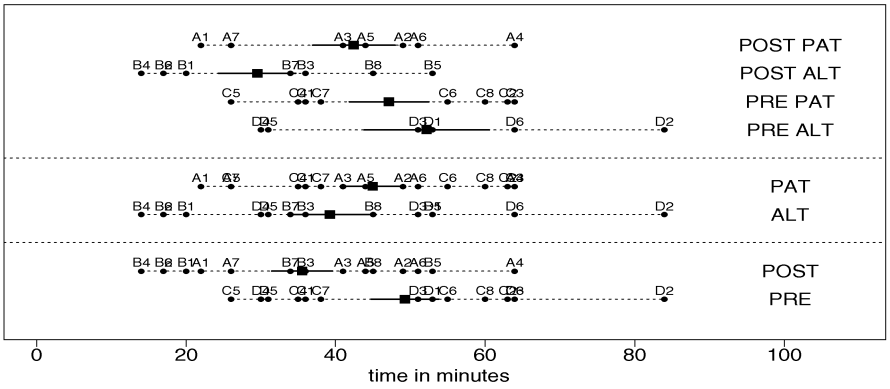


Fig. 4. Time required for program *Boolean Formulas* work task 1.

### 3.3 Decorator: Communication Channels (CO)

**Program description**. *Communication Channels* is a wrapper library. A communication channel establishes a connection for transparently transferring arbitrary-length packets of data and one can turn on additional logging, data compression, and encryption functionality. The library does not implement the functionality itself, but only provides a FACADE to a system library. However, this application of the FACADE pattern is irrelevant to the experiment.

The PAT version is designed with a DECORATOR for adding the functionality to a bare channel, having the classes for logging, data compression, and encryption as decorator classes. The program consists of 365 lines in six classes.

The ALT version comprises but a single class, that uses flags and if-sequences for turning functionality on or off; the flags can be set when creating a channel. It consists of 318 lines. Communication channels is the only program where the ALT program has a structured (as opposed to object-oriented) design.

#### 3.3.1 Work Task 1

*"Enhance the functionality of the program such that error-correcting encoding (bit redundancy) can be added to communication channels."* The underlying functionality is again provided by a given class, so the subjects only had to integrate the new functionality into the program.

The PAT subjects had to add a new DECORATOR class while the ALT subjects had to make additions and changes at various points in the existing program.

**Expectations**. We expect two influences of the DECORATOR on the subjects' behavior. First, the ALT version is easier to understand because its behavior is not delocalized as in the multiple decorator classes. This would lead to the conclusion that the ALT groups are faster than the PAT groups, especially in the pretest. Second, a counter-influence results from the structure of the DECORATOR: The functionality is encapsulated in classes and one need hardly care about mutual influences. In particular, in the ALT version, the subjects have to ensure they add the new functionality at the correct places in the program for proper sequencing of the various switchable functionalities; this will consume time and may lead to omissions and mistakes. We expect the second influence to be stronger than the first and, hence, the PAT version to be preferable (E1), especially at higher levels of pattern knowledge (E2).

**Results**. As one can see from Fig. 5 the PAT groups are indeed significantly faster than ALT groups (38 percent faster, 28.8 minutes versus 46.2 minutes, significance $p < 0.001$), confirming E1. The pattern-solution is clearly preferable.

There is no significant difference between PRE-ALT and POST-ALT, as expected (46.5 minutes versus 45.9 minutes, significance $p = 0.46$), but also none between PRE-PAT and POST-PAT (27.5 minutes versus 29.8 minutes, significance $p = 0.29$), thus rejecting E2. This means the positive effect of pattern use is even independent of pattern knowledge in this case.

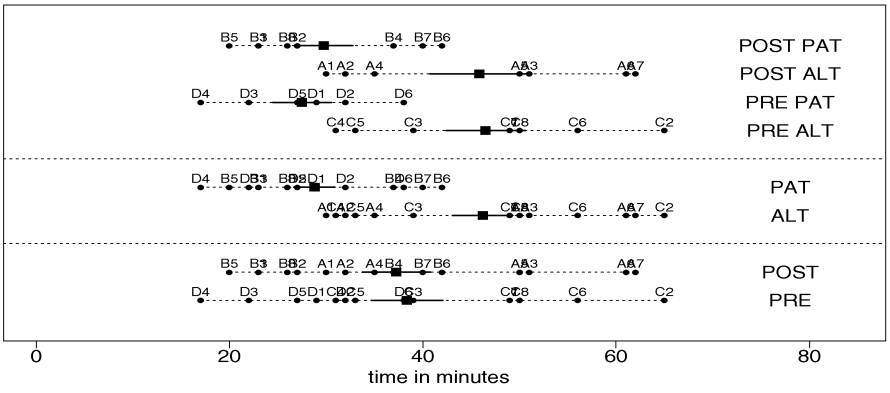


Fig. 5. Time required for program *Communication Channels* work task 1.

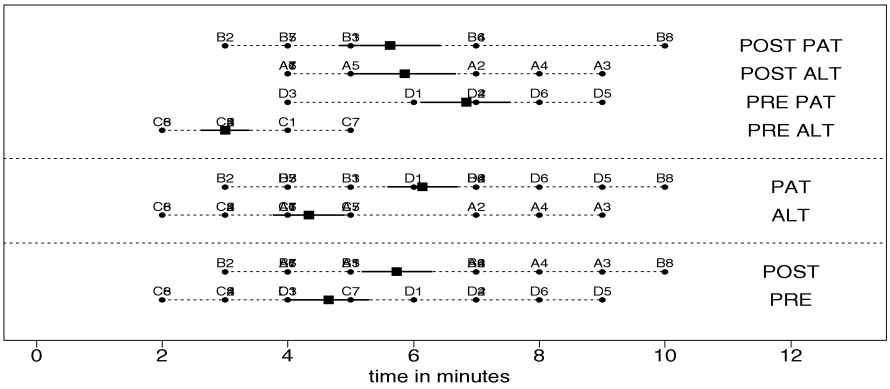


Fig. 6. Time required for program *Communication Channels* work task 2.

The pattern-solution is also superior in terms of correctness: Errors were made by seven out of eight PRE-ALT subjects and by six out of seven POST-ALT while in the PAT group no errors occurred at all.

### 3.3.2 Work Task 2

A communication channel has different states (namely, opened, closed, or failed) and its operations have different result codes (OK, failure, or impossible). Work task 2 called to "*determine under which conditions a reset() call will return the 'impossible' result.*" To do this the subjects had to find the spots where the states were changed. In the PAT version, these spots are spread over the different decorator classes.

**Expectations**. Program understanding is gained in the first working task. So, only the new details relevant for this task need to be understood now. This will be easier for the more localized ALT program with respect to both work time (E3) and correctness (E4).

**Results**. The results as shown in Fig. 6 are inconclusive for this task: The error rate in the ALT groups is almost as high as in the PAT groups (in contrast to E4) and the ALT group is much faster in the pretest than in the posttest. The latter is unexpected and can only be explained by a subject fatigue effect in the (afternoon) posttest or by pure chance (which is plausible since the task is only five minutes long). Overall, E3 is still supported though. Note that this task is rather minor and that all figures use different scales.

### 3.3.3 Work Task 3

"*Create a channel object that performs compression and encryption.*" The ALT subjects had to create only a single object, giving parameters for the functionality flags, while PAT subjects had to determine the proper nesting of the decorators to get the required functionality in the requested order. (A similar sequence problem plagued the ALT subjects in task 1.)

**Expectations**. The PAT groups will take longer (E5) and commit more errors (E6).

**Results**. Both expectations are supported (see Fig. 7): Overall, the ALT group is significantly faster (53 percent faster, three minutes versus 6.4 minutes, significance $p < 0.001$) than the PAT group. More importantly, we counted six wrong solutions (out of 14) for PAT, while no errors were observed for ALT. However, this object creation problem could be overcome by a suitable convenience method without changing the overall design.

## 3.4 Composite and Abstract Factory: Graphics Library (GR)

**Program description**. *Graphics Library* contains a library for creating, manipulating, and drawing simple types of graphical objects (lines and circles) on different types of graphical output devices (alphanumeric display, pixel display). In a central class (generator), the output device is selected. Depending on the output device, the corresponding types of graphical objects are created. Some basic

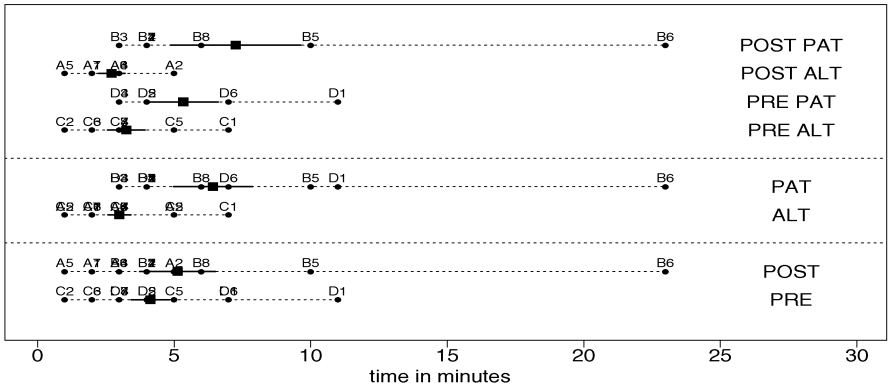


Fig. 7. Time required for program *Communication Channels* work task 3.

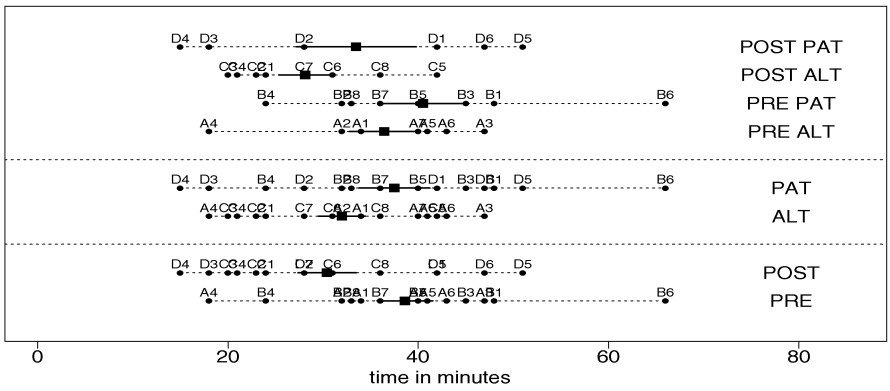


Fig. 8. Time required for program *Graphics Library* work task 1.

graphical objects (lines and points) are implemented identically for all graphical output devices, but the implementation of complex objects like circles or the graphical context depends on the graphical output device. Furthermore, graphical objects can be collected in groups, that can be manipulated like individual objects.

Patterns used in the PAT version of this program are ABSTRACT FACTORY for the generator classes and COMPOSITE for hierarchical object grouping.

The ALT version of the program realizes the instantiation of the appropriate classes for each graphical output device by switch-statements in but a single generator class. The combination and manipulation of graphical object groups are realized with a quasi-COMPOSITE. The only difference is that groups are not treated as graphical objects as in the COMPOSITE. As a result, a group B is included in another group A by adding each element of B individually to A, i.e., there is no hierarchical group nesting.

This program pair has the smallest structural difference between the PAT and ALT version of all four program pairs in the experiment. The PAT version is 13 classes in 682 lines; the ALT version 11 classes in 663 lines.

### 3.4.1 Work Task 1

*"Add a third type of output device (plotter)."* Subjects maintaining the PAT program had to introduce a new concrete factory class, extend the factory selector method, and add two concrete product classes. Subjects in the ALT groups had to enhance the switch statements in all methods of the generator class. The appropriate classes of graphical objects for the new output device had to be added as for PAT.

**Expectations**. Regarding the maintenance task, the time for finding the changes and additions is expected to be almost equal for the PAT and the ALT groups. So, the main difference in time required for this task will be caused by program understanding. Here, we expect the simpler ALT program to be easier to understand, at least in the pretest (E1).

Pattern knowledge will help both groups (E2) because of the COMPOSITE structure in both programs. The pattern group may profit a little more from the pattern course because it eases understanding the structure of the ABSTRACT FACTORY.

**Results**. The results shown in Fig. 8 support both expectations. Both groups maintaining the ALT program were faster than the corresponding PAT groups with the same pattern knowledge level, supporting E1 (15 percent faster, 32 minutes versus 37.5 minutes, total significance $p = 0.10$). The improvement from PRE to POST (E2) is 17.3 percent (40.5 minutes versus 33.5 minutes, significance $p = 0.17$) for the PAT group and 22.8 percent (36.4 minutes versus 28.1 minutes, significance $p = 0.031$) for the ALT group. That is 21.2 percent overall (38.6 minutes versus 30.4 minutes, significance $p = 0.021$).

### 3.4.2 Work Task 2

Determine whether or not a certain sequence of operations would result in an x-shaped figure. This work task is a small comprehension test concerning the COMPOSITE structure. The key to the answer for both groups is finding out that only references to graphical objects (not copies of objects) are stored in an object group.

**Expectations**. The structure of both programs is quite similar in the region of interest. So, we do not expect to observe significant differences between the ALT and the PAT groups (E3). But, we expect a difference between PRE and POST: Subjects without pattern knowledge are slower than subjects with pattern knowledge (E4) because the latter are familiar with the COMPOSITE.

**Results**. As we see in the lower part of Fig. 9, the difference between PAT and ALT (PAT is 21 percent faster than ALT, 13.6 minutes versus 17.2 minutes, significance $p = 0.085$) is very similar to the difference between POST and PRE (POST is 21 percent faster than PRE, 13.6 minutes versus 17.2 minutes, significance $p = 0.091$). Both are only weakly significant. However, we tend to consider the large value of subject A6 an outlier. Deleting it makes both differences disappear, so that E3 is confirmed but E4 is rejected: The performance does not depend on pattern knowledge.

## 4 CONCLUSION

We investigated the question whether (with respect to maintenance) it is useful to design programs with design patterns even if the actual design problem is simpler than that solved by the design patterns, i.e., whether using patterns that over-kill the problem at hand is useful or harmful.

We found evidence of both cases, depending on the situation. Software engineering common sense turned out

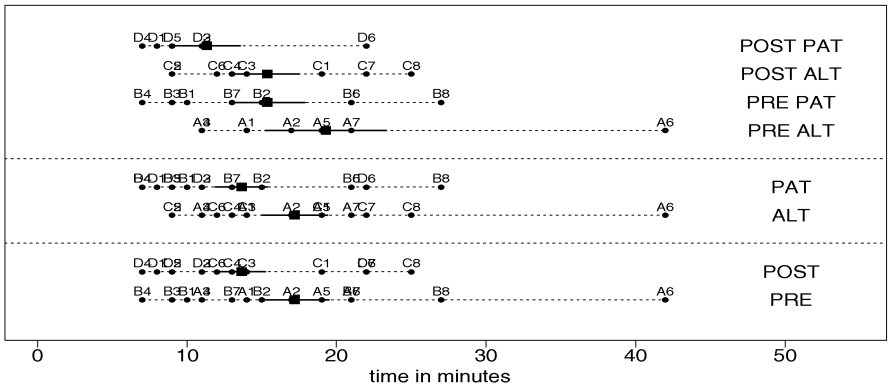


Fig. 9. Time required for program *Graphics Library* work task 2.

to be a pretty accurate (but not perfect) predictor of these effects for three out of the four programs used in this experiment. Summarizing the individual expectations versus actual results for these programs and tasks yields the following picture:

- Program "Stock Ticker (ST)" (OBSERVER):
  **Expectation**. The pattern solution is more complicated and, thus, harmful, unless its flexibility is really required.
  **Actual result**. A negative effect from unnecessary application of the OBSERVER pattern, particularly for subjects with low pattern knowledge.
- Program "Boolean Formulas (BO)" (COMPOSITE, VISITOR):
  **Expectation**. The VISITOR is difficult to understand and, thus, harmful.
  **Actual result**. A neutral effect—the VISITOR does not significantly increase the required time.
- Program "Communication Channels (CO)" (DECORATOR):
  **Expectation**. Due to the isolation of different parts of the functionality (and, thus, delocalization of the overall functionality), the pattern solution is easier to change, but more error-prone to call.
  **Actual result**. As expected.
- Program "Graphics Library (GR)" (COMPOSITE, ABSTRACT FACTORY):
  **Expectation**. The two versions are structurally similar, so we anticipate at most small performance differences.
  **Actual result**. As expected, only small differences were found.

Note that these expectations were only qualitative, so the quantitative experiment results provide additional information beyond confirming or rejecting the hypotheses.

We suggest the following lessons learned. First, it is usually, but not always, useful to use a design pattern if there are simpler alternatives. Second, use software engineering common sense to find the exceptions where a simpler solution should be prefered, even if a design pattern solution could easily be applied. Third, even where this common sense suggests that using a pattern might not be a good idea, it is sometimes right to use it (as with the Visitor in our program BO). Hence, if in doubt, using the pattern rather than the simpler solution appears to be a good default approach. Fourth, a thorough understanding of specific design patterns often helps when maintaining programs using them, *even if* these programs are neither very large nor very complicated. If this observation holds in general, it suggests limitations to the usefulness of patterns once the catalog of available patterns becomes large and programmers do not know them all.

We emphasize that, unless there is a clear reason to prefer the simpler solution, it is probably wise to choose the flexibility provided by the design pattern solution because unexpected new requirements often occur. This aspect was deliberately ignored in our experiment. In those cases where the pattern solution was not beneficial, its added complexity can be viewed as the price for its flexibility.

As a consequence of lessons one and three, we need to make sure that the software engineers are familiar with alternatives. This means that in our university courses, we must not just teach the current fad blindly (whether it is OO or patterns), but we should teach alternative approaches as well.

Further research should address the following questions: Are there alternative simpler solutions for specialized applications of other (kinds of) design patterns as well? Are the trade-offs involved similar to the ones discussed here? What are the effects of pattern versus nonpattern designs for *long term* maintenance involving many interacting changes? How does the use or nonuse of patterns influence activities other than pure maintenance, e.g., inspections or code reuse? Can we characterize the situations in which (current) design common sense misleads us?

## ACKNOWLEDGMENTS

The authors would like to thank Ernst Denert and SD & M for making the experiment possible and all their subjects for being so interested in it. They also thank the second reviewer of their rejected Foundations of Software Engineering submission for the important remark regarding university education.

## REFERENCES

[1] K. Beck, J.O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlissides, "Industrial Experience with Design Patterns," *Proc. 18th Int'l Conf. Software Eng.*, pp. 103–114, Mar. 1996.

[2] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu, "Automatic Code Generation from Design Patterns," *IBM Systems J.*, vol. 35, no. 2, pp. 151–171 1996.
[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns.* Chichester, UK: John Wiley and Sons, 1996.
[4] L.B. Christensen, *Experimental Methodology,* sixth ed. Heights, Mass.: Allyn and Bacon, Needham, 1994.
[5] B. Efron and R. Tibshirani, *An Introduction to the Bootstrap,* Monographs on Statistics and Applied Probability 57. New York, London: Chapman and Hall, 1993.
[6] G. Florijn, M. Meijers, and P. vanWinsen, "Tool Support for Object-Oriented Patterns," *Proc. 11th European Conf. Object-Oriented Programming (ECOOP),* M. Aksit, ed., pp. 472–495, June 1997.
[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1995.
[8] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Software Eng.,* vol. 12, no. 1, pp. 96–109, Jan. 1986.
[9] L. Prechelt, "An Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation," Technical Report 9/1997, Fakultät für Informatik, Universität Karlsruhe, Germany, June 1997, ftp.ira.uka.de.
[10] L. Prechelt, B. Unger, M. Philippsen, and W.F. Tichy, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Information During Program Maintenance," to appear in *IEEE Trans. Software Eng.,* http://www.ipd.ira.uka.de/~prechelt/Biblio/.
[11] L. Prechelt, B. Unger, and D. Schmidt, "Replication of the First Controlled Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation," Technical Report wucs-97-34, Washington Univ., Dept. of Computer Science, St. Louis, Mo., Dec. 1997, http://www.cs.wustl.edu/cs/cs/publications.html.

**Lutz Prechelt** worked as a senior researcher at the School of Informatics, University of Karlsruhe, where he received the diploma (1990) and the PhD degree (1995) in informatics. His research interests include software engineering (in particular, using an empirical research approach, about which he also authored a book), compiler construction for parallel machines, measurement and benchmarking issues, and research methodology. Since April 2000, he is head of quality assurance at abaXX Technology, Stuttgart. Prechelt is a member of the IEEE Computer Society, the ACM, and the German Informatics Society.

**Barbara Unger** graduated with a diploma degree in 1995 and received the PhD degree in 2000 from the University of Karlsruhe. Her favorite research interests are in empirical software engineering with a main focus on design patterns and team communication. She joined sd&m, Frankfurt, in May 2001. She is a member of the IEEE and the IEEE Computer Society.

**Walter F. Tichy** received the BS degree from the Technical University in Munich in 1974 and the MS and PhD degrees in computer science from Carnegie-Mellon University in 1976 and 1980, respectively. He is professor of computer science at the University Karlsruhe, Germany. He is also the director of the Software Engineering Department, including a SUN authorized Java Center, at Forschungszentrum Informatik, a research and transfer institute associated with the University. Previously, he was a senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and on the faculty of the Computer Science Department at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. His current research projects include experimental methods in computer science and software engineering, software architecture and pattern research, software configuration management, cluster computing, compilers and programming environments for parallel machines, and opto-electronic interconnects. He has consulted widely for industry. He is a member of the ACM, the German Informatics Society, and the IEEE.

**Peter Brössler** (no bio available)

**Lawrence G. Votta** received the BS degree in physics from the University of Maryland, College Park, Maryland in 1973, and the PhD degree in physics from the Massachusetts Institute of Technology, Cambridge, Massachusetts in 1979. He currently leads the performance and availability modeling and analysis group of the Common Platform Development Department in Motorola's Network Systems Sector. His research interests are high availability computing (new) and empirical software engineering (his old favorite). He has authored or coauthored more than 40 articles and book chapters on software engineering including empirical studies of software development from highly controlled experiments investigating the best methods for design reviews and code inspection to anecdotal studies of a developer's time usage in a large software development. He is a member of the IEEE and the ACM and is currently serving as an associate editor of the *IEEE Transactions on Software Engineering*.