A controlled experiment in maintenance comparing design patterns to simpler solutions — D R A F T —

Lutz Prechelt (prechelt@ira.uka.de)
Barbara Unger (unger@ira.uka.de)
Walter F. Tichy (tichy@ira.uka.de)
Fakultät für Informatik, Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/608-7343

Peter Brössler
(Peter.Broessler@sdm.de)
sd&m GmbH & Co KG
Thomas-Dehler-Str. 27
80737 München, Germany

March 9, 1998

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 to simpler alternative designs. The subjects were professional software engineers.

We find that programs using design patterns can be more difficult to maintain than more conventional solutions, or can be simpler to maintain (even where the specific core advantages of the pattern solution are not relevant), or can be neutral. Carefully applied common sense is a good discriminator between these cases. One should consider alternatives and rely on judgment even when design patterns are applicable.

1 Design patterns and flexibility

Object-oriented design patterns in the sense of Gamma et al. [5] 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. [5] aim at reducing coupling and increasing flexibility within systems. For instance, many of these patterns delay decisions until run time 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 being proven solutions, using patterns purportedly provides additional advantages: Design patterns provide precise terminology that improves communication among designers [1] or from designers to maintainers [5, page 2]. They purportedly also make it easier to think clearly about a design and encourage the use of "best practices".

Given the popularity of (in particular) the Gamma et al. design patterns, one can expect that they will often be used in situations where

not their whole flexibility is needed: The pattern solves the problem at hand, but is more powerful than required.

In such situations there will be two competing forces: On the one hand, applying the pattern might be a good idea because of the abovementioned pattern advantages of common terminology, proven solutions, and best practices. On the other hand, it may be a bad idea because the solution applied is more complicated than necessary and may thus make understanding and change more difficult. For instance, [1] warns that "a design pattern is not a rule to be followed blindly". Another aspect of this tradeoff is delibarately ignored in the present experiment: Even if the pattern solution is overly powerful right now, its flexibility may be needed later on, even if this is not currently expected.

1.1 Experiment overview

The controlled experiment described herein assesses design patterns versus alternative designs in the context of program maintenance. We consider four different programs with different design patterns. Among the several flexibility and functionality properties of the design pattern solution of each program at least one is not actually required for the original program and for the given maintenance tasks analyzed in the experiment. 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 in a standard fashion whereas version ALT employs a simpler solution, which does exhibit only the functionality and flexibility actually required.

The programs are well-documented, modestly-sized, artificial programs; they contain implementations of the design patterns Abstract Factory, Composite, Decorator, Facade, Observer, and Visitor as described in the book [5]. The subjects are professional software engineers.

1.2 Structure of this article

We now shortly discuss related work and then, in Section 3, summarize the experiment design and conduct, including a short statement of the experiment objective, a description of the subjects' background, the division into groups, the environment, the measurement, and discussion of the internal and external validity of the experiment. Section 4 describes the programs used in the experiment, the work tasks, the expectations, and the actual results. We will assume the reader understands the relevant design patterns and their properties; we will thus not describe them in detail. The conclusion sketches the common denominator of the results, possible consequences for proper program design, and further research questions to be investigated.

2 Related work

A lot of work is currently being done in both scientific and industrial context towards identifying design patterns, writing them up, teaching them, using them, building support tools, etc. [1, 2, 3, 4]. Reports on the effects of patterns are available in anecdotal form from various practitioners [1], but there is little work done yet in a quantitative fashion, let alone in a controlled environment.

In fact the only quantitative, controlled experiment on patterns reported so far seems to be [7] (see [6, 8] for details). It investigates communication improvements through patterns in a maintenance situation. Maintenance can be done quicker and with fewer errors for software that documents its use of design patterns explicitly as opposed to the same software without the pattern documentation. This result confirms some of the purported positive effects on communication but does not describe effects of patterns on actual software structure.

3 Description of the experiment

3.1 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 properties of a particular design are 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, harmful, or neutral. In detail the objectives of the experiment were:

- to test whether using design patterns in programs is advantageous even if the flexibility provided by the design pattern is not required;
- to test whether the above effect depends on the particular program (or pattern);
- to test how the above effect depends on the amount of pattern knowledge of the software engineers.

3.2 Design

Our experiment design uses three independent and two dependent variables. The independent variables are the program (and change tasks), the program version, and the amount of pattern knowledge; the dependent variables are cost and correctness.

- Independent variable "program and change tasks" (4 values): We use four different programs. Each has a different purpose, uses different design patterns, and involves very different maintenance tasks. Both, programs and tasks, will be described in detail in Section 4.
- Independent variable "program version" (binary): 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.
- Independent variable "amount of pattern knowledge" (binary): This is the difference between pretest and posttest. The experiment had two parts on two different days. The first part (the pretest) was performed in the morning of the first day. Then a pattern course was taught during the rest of the day and the morning of the next day. In the afternoon of day 2 the second part of the experiment (the posttest) 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, although we cannot quantify the knowledge levels or their difference.
- Dependent variable "cost" (continuous): The maintenance cost is represented by the time (in minutes) taken for each maintenance task. Subsequently we will always call this variable "time".
- Dependent variable "correctness" (binary): We decided whether the participants' solutions fulfilled the requirements of the task or not. Minor defects were ignored. 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. The design is summarized in Table 1.

3.3 Subjects and groups

The 29 subjects are all professional software engineers. On average, they had worked as

	group A	group B	group C	group D
1st problem	ST Pat	GR Pat	CO Alt	BO Alt
2nd problem	GR Alt	ST Alt	BO Pat	CO Pat
pattern course				
3rd problem	CO Alt	BO Alt	ST Pat	GR Pat
4th problem	BO Pat	CO Pat	GR Alt	ST Alt

Table 1: Order of programs per group. ST, BO, CO, and GR are the programs and Alt or Pat indicates which program version was used. See descriptions in the text.

software professionals for 4.1 years and their C++ experience was 2.4 years and 21.1 KLOC. This is how, on average, they claimed they spend their working time: 14.8% requirements analysis, 15.3% design of new software, 17.9% coding new software, 6.8% design for maintenance tasks, 11.4% coding for maintenance tasks, 18.1% testing.

15 subjects had previous pattern knowledge, 13 of them on at least one of the patterns used in the experiment.

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 good 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 actual participant submitted his questionnaire only when the experiment started and was assigned ad-hoc. The resulting actual group sizes were 6 to 8 subjects in each group, with 3 to 5 having theoretical or practical pattern knowledge before the course.

3.4 Experiment conduct

The experiment was performed in November 1997 in a software company located in Munich. Both, course and experiment, were performed in a conference room. The pretest started at 9:30h in the morning, the posttest at 12:40h the next day. Both parts lasted approximately 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. Subjects had to fill in line numbers of the program listing where changes or additions were required and they had to specify what to add or change. If classes or methods had to be added the subjects were only asked to specify the signature of the method or class, not its body. Still, though, almost all of the solutions contained method bodies as well.

At various points during the experiment, in particular immediately before and after each task, the subjects recorded the time of day. From this information we computed the time required for each subtask.

Each subject worked in his or her own pace. Although the overall time for the experiment was limited, all participants were able to finish in the allotted time.

3.5 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, based on the substitute measures of capability available from the pre-experiment questionnaire, as described in Section 3.3. We feel the resulting groups were

	group A	group B	group C	group D
group size	7	8	8	6
pattern knowledge	3	4	5	3

Table 2: Group size. First row: Group size overall. Second row: Number of subjects having pattern experience before the pattern course.

reasonably similar and our results give no reason to believe the opposite.

A second consideration is the precision and accuracy of the time stamps recorded by the subjects. We found these data to be highly accurate and reliable: Of over 600 time stamps, only a single one was missing and about a dozen were found to be incorrect. All of them could be detected and reconstructed based on internal consistency within the large number of interdependent timestamps provided.

3.6 Threats to external validity

External validity is the degree to which the results are generalizable, i.e., transfer to other situations. There are several sources of differences between the experimental and real software maintenance situations that limit the generalizability of this experiment: First, the original designers and implementors maybe 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 *practical* pattern knowledge, compared to the theoretical knowledge of many of 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), which will typically trade some of the incorrectness observed in the experiment against additional time. Furthermore, it is unclear in which manner the results obtained here transfer to other design patterns and their alternatives.

4 Results

It turns out that the effects of PAT versus ALT and of pretest versus posttest are very different for the four different programs. Hence the results are discussed program-by-program. For each of the programs we describe

- the size and functionality of the program,
- the use of design patterns,
- the work tasks and the solutions,
- our expectations for each work task,
- and the actual results.

The expectations represent a common-sense judgement of whether and why the PAT or the ALT version may be better for this particular task (such expectations are indexed $_P$, for 'PAT vs. ALT') and/or what influence the level of pattern knowledge may have (i.e., comparing pretest to posttest, such expectations are indexed $_k$, for 'knowledge'). The expectations form the basis of our discussion and interpretation of the quantitative results. Statistical tests were performed using the percentile method (one-sided) after 10000 iterations of bootstrap resampling of the difference of arithmetic means.

The original documents including the programs and the work tasks are available at `http://wwwipd.ira.uka.de/~unger/exp/Munich/materials.html`

4.1 Observer: Stock Ticker (ST)

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

Both versions of *Stock Ticker* consist of seven classes. The PAT version contains an OBSERVER. The displays are implemented as observer classes and must register at the OBSERVER's subject once; the subject notifies the displays each time the data have changed and the displays fetch the new data on notification. Four of the seven classes are employed in the pattern (the subject providing the data and registering the displays, the superclass of the displays, and two concrete displays.) This version of the program has 343 lines of code where lines of code is defined as all 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 external generation and dynamic registration of observers is implemented. This version has 279 lines.

4.1.1 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 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. Once this understanding is achieved, the task becomes small and simple.

Expectations: We expected that most of the time is required for understanding the program structure. The structure of the PAT version appears to be more complicated than the structure of the ALT program version. When subjects do not have knowledge about the OBSERVER pattern they have to find out how the OBSERVER mechanism works. In this case subjects probably require more time than for the ALT version. Given sufficient pattern knowledge, on the other hand, the PAT group may understand the program structure more quickly than the ALT group. With these considerations we pose the following expectations for this task:

> **Expectation ST1$_{P,k}$:** Groups without pattern knowledge are slower for the PAT program than for the ALT program.
>
> **Expectation ST2$_{P,k}$:** Groups with sufficient pattern knowledge are faster for the PAT program than for the ALT program.

Results: Figure 1 supports expectation ST1$_{P,k}$: pretest subjects require more time (2.5 times more!) if they are working on the PAT version (significance $p = 0.000$). Expectation ST2$_{P,k}$ is not supported: Even in the posttest the PAT subjects required more time than the ALT subjects ($p = 0.023$), although they were twice as fast as in the pretest. We conclude that for this application and this type of maintenance tasks the use of the OBSERVER pattern is harmful, in particular if no pattern knowledge is present.

4.1.2 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 in that it requires something that the PAT version already provides, but the ALT version does not. Therefore, we expect the ALT version to

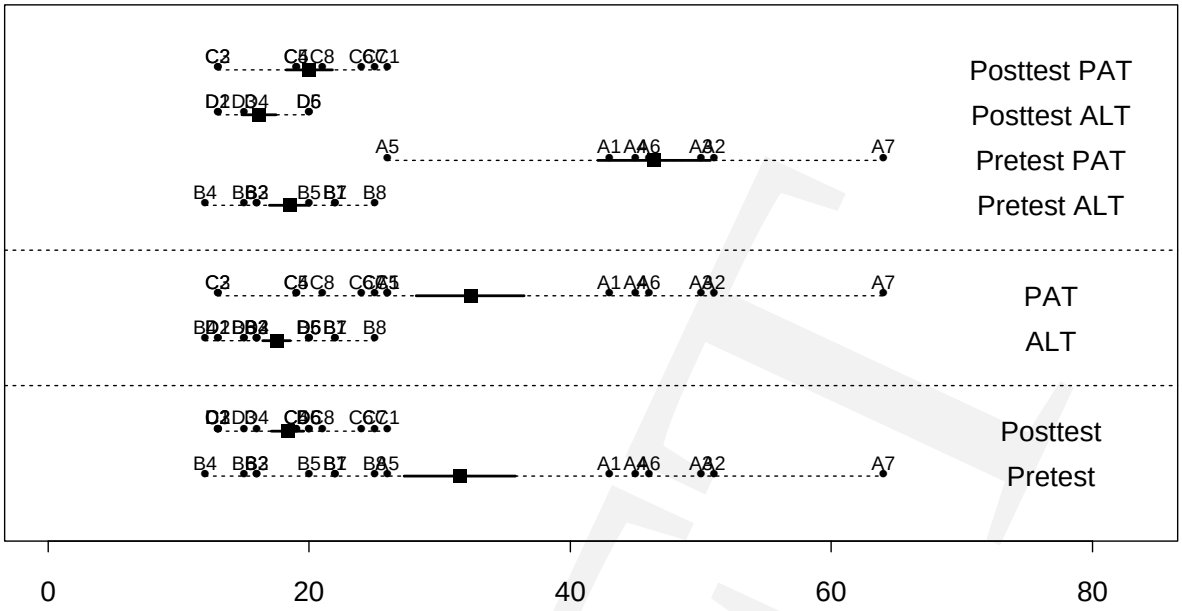

Figure 1: Time required for program *Stock Ticker* work task 1. Each dot marks one subject, the square is the arithmetic mean, the line indicates plus/minus one standard error of the mean. The top area (4 lines) shows the four individual groups. The mid area (2 lines) shows the same but with the pretest and posttest groups of PAT combined and the pretest and posttest groups of ALTcombined. Likewise, the bottom area (2 lines) shows PAT and ALT groups combined.

be clearly at disadvantage. In the PAT version of the program the subjects need to know that the OBSERVER already implements the functionality required. In the posttest the subjects ought to have this knowledge; in the pretest it might be missing. In the ALT version the subjects may have to re-invent the OBSERVER solution (in the pretest) and in any case have to implement it. This leads to the expectation that:

> **Expectation ST3$_P$:** The ALT version requires much more time for this maintenance task than the PAT version.
>
> **Expectation ST4$_k$:** With pattern knowledge both versions of the program can be maintained faster than without.

Results: Expectation ST3$_P$ is supported by the data in Figure 2. The unfair task is completed on average 29% faster on the PAT version (significance of difference $p = 0.045$). Expectation ST4$_k$ is not confirmed by the results. the ALT group was even slightly slower in the posttest than in the pretest. There are three possible explanations: First the ALT subjects may have expected a pattern-relevant exercise after the course and they may have spent time searching for a pattern. Second, the posttest was performed in the afternoon after 4 hours of pattern course in the morning whereas the pretest was performed in the morning. So some subjects may have had difficulty concentrating. The PAT posttest group was faster than the pretest group, as expected, but the difference is not significant ($p = 0.217$).

4.2 Composite and Visitor: Boolean Formulas (BO)

Program description: The program *Boolean Formulas* (or BO, for short) contains a library for representing boolean formulas (composed from n-ary AND, n-ary OR, binary XOR, unary NOT, and variables) and for printing the formulas in two different styles. Furthermore, it contains a small main program which gener-

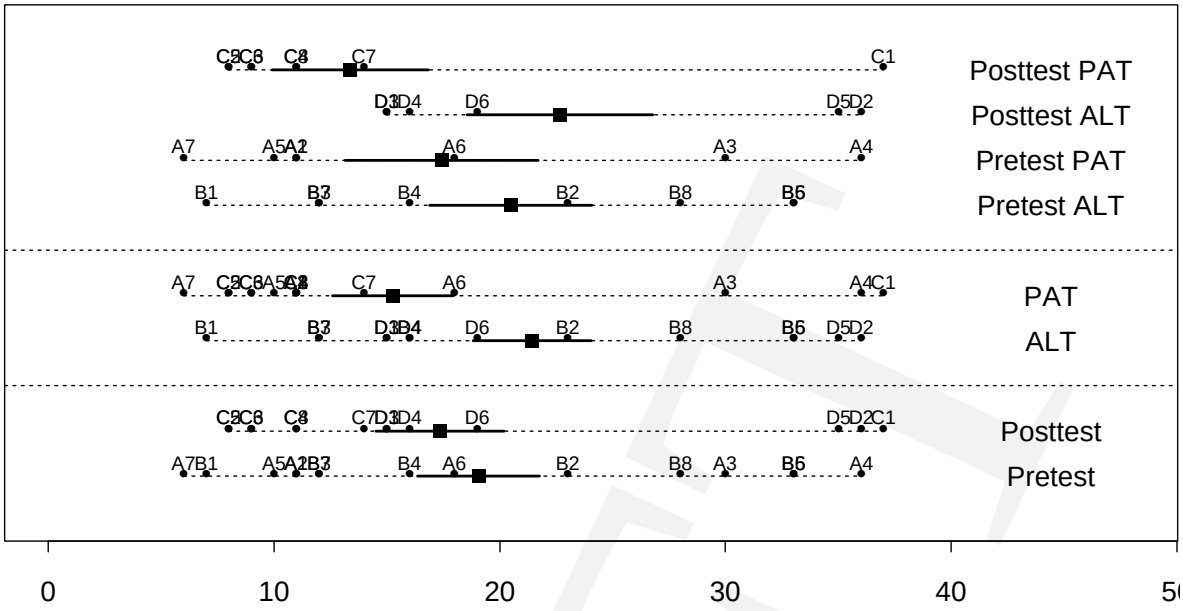

Figure 2: Time required for program *stock ticker* work task 2.

ates 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 where the AND, OR, XOR, and NOT classes are composites (containers) and the variable classes are leafs. The printing routines are implemented as VISITORs, inheriting from an abstract superclass. 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: 8 classes spanning 374 lines. It has almost the same structure as the PAT version except for the VISITOR pattern, which is completely missing. The different printing routines are located directly in each COMPOSITE class instead. The real-world advantage of the VISITOR solution is its flexibility for adding new visitors without changing the COMPOSITE classes.

4.2.1 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: It might in principle be easier to create a new class similar to another rather than adding methods to a bunch of classes, which would prefer the PAT groups, but the VISITOR pattern is technically quite difficult to understand. So we expect that it will take some more time for the PAT groups to understand the VISITOR pattern than for the ALT groups to find where to add the methods. Gaining pattern knowledge should help all groups because even in the ALT program a COMPOSITE is present, so in the posttest the subjects presumably understand the structure of the formula representation faster. The PAT group might profit more from the pattern course than the ALT group because the working mechanism of the VISITOR is particularly confusing beforehand.

Expectation $BO1_P$: The PAT groups will be slower than the ALT groups.

Expectation $BO2_k$: Increased pat-

tern knowledge helps to understand both versions faster.

Expectation BO3$_k$: The PAT group will profit more from the pattern course than the ALT group.

Results: As one can see from Figure 3, expectation BO1$_P$ is supported by the posttest, but not by the pretest: In the pretest, the PAT group is slightly (but nonsignificantly) faster than the ALT group ($p = 0.299$). In the posttest the ALT group is faster than the PAT group, as expected ($p = 0.034$). This is an inconclusive result. Probably the expectation is violated because in the pretest the VISITOR is largely just taken for granted and imitated by the subjects (instead of analyzed and understood) and thus does not increase complexity for the PAT group, but still exhibits its advantage of centralization. The same reasoning may apply to BO3$_k$, which is also not supported. Taken together, these explanations mean that an unrequired VISITOR, although it appears complicated, is not necessarily harmful.

Expectation BO2$_k$ is supported by the data analysis: The posttest groups are significantly faster (mean difference: 28 percent, $p = 0.034$). The ALT posttest group (with only the COMPOSITE pattern) requires 43 percent less time than the corresponding pretest group (significance $p = 0.004$). An improvement in the PAT groups is also present, but not significant (10 percent, $p = 0.26$). This rejects expectation BO3$_k$.

4.2.2 Work task 2

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

4.3 Decorator: Communication Channels (CO)

Program description: *Communication Channels* (or CO, for short) is a program for configuring and running a packet-switched data transmission line. Such a 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, instead it calls a system library providing the functionality and implements only a unified interface for the combination of the parts. Both versions of the program thus realize a FACADE pattern, but this is irrelevant for the experiment.

The PAT version is designed with a DECORATOR for adding the functionality to a bare channel. The bare channel is a component class, the classes for logging, data compression, and encryption are decorator classes.

The ALT version comprises but a single class, which uses flags and if-sequences for turning functionality on or off; the flags can be set when creating a channel. The PAT program consists of 365 lines in six classes and the ALT program version consists of 318 lines in only one class. Communication channels is the only program where the ALT program has a modular (as opposed to object-oriented) design.

4.3.1 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.

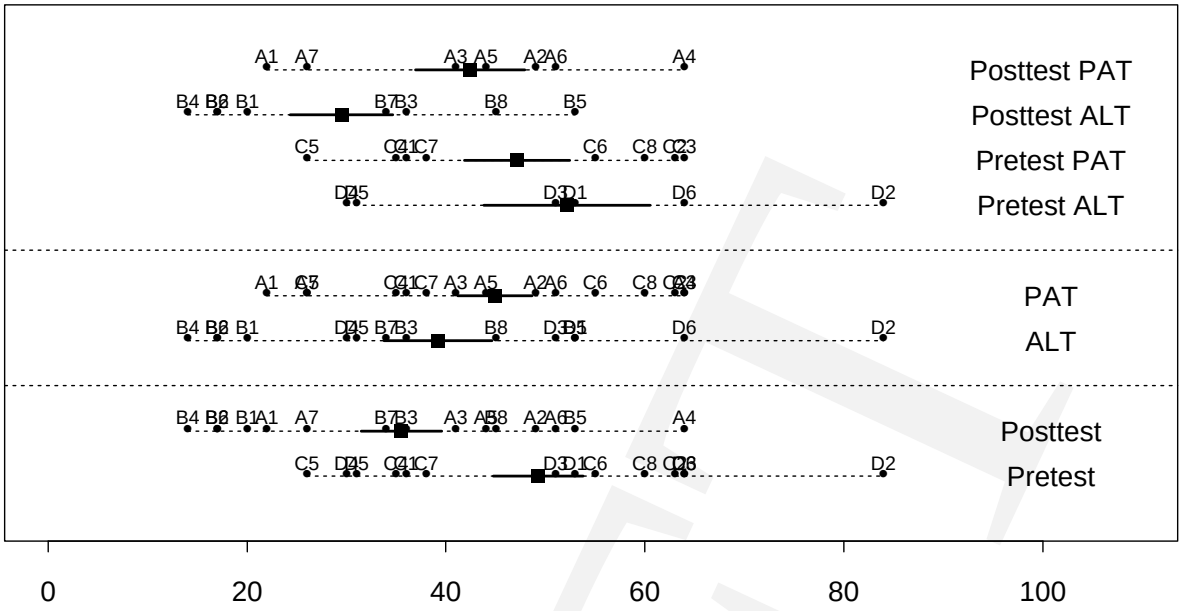

Figure 3: Time required for program *Boolean Formulas* work task 1.

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. The second 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 may not only consume time but also lead to omissions and mistakes. We expect the second influence to be stronger than the first, especially at higher levels of pattern knowledge.

> **Expectation CO1$_P$:** Subjects working on the PAT version are faster than subjects working on the ALT program version.
>
> **Expectation CO2$_P$:** Working on the ALT program is more error-prone.
>
> **Expectation CO3$_{P,k}$:** On a PAT program subjects become faster with more pattern knowledge. For the ALT program there is no difference.

Results: As one can see from Figure 4 the PAT groups are indeed significantly faster than ALT groups, supporting CO-PA ($p = 0.000$). The pattern-solution is clearly preferable.

Expectation CO3$_{P,k}$ is not confirmed. There is hardly any difference between pretest and posttest for the ALT groups, as expected ($p = 0.46$), but also none for the PAT groups ($p = 0.29$). This means the positive effect of pattern use is even independent of pattern knowledge in this case!

The pattern-solution is not only superior in terms of speed, but also in terms of correctness, supporting CO2$_P$: The number of errors made by the subjects is much higher in the ALT groups: Errors were made by 7 out of 8 subjects in the pretest and by 6 out of 7 subjects in the posttest while in the PAT group no errors occurred at all.

4.3.2 Work task 2

A communication channel has different states (opened, closed, failed) and its operations have different result codes (OK, failure, 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.

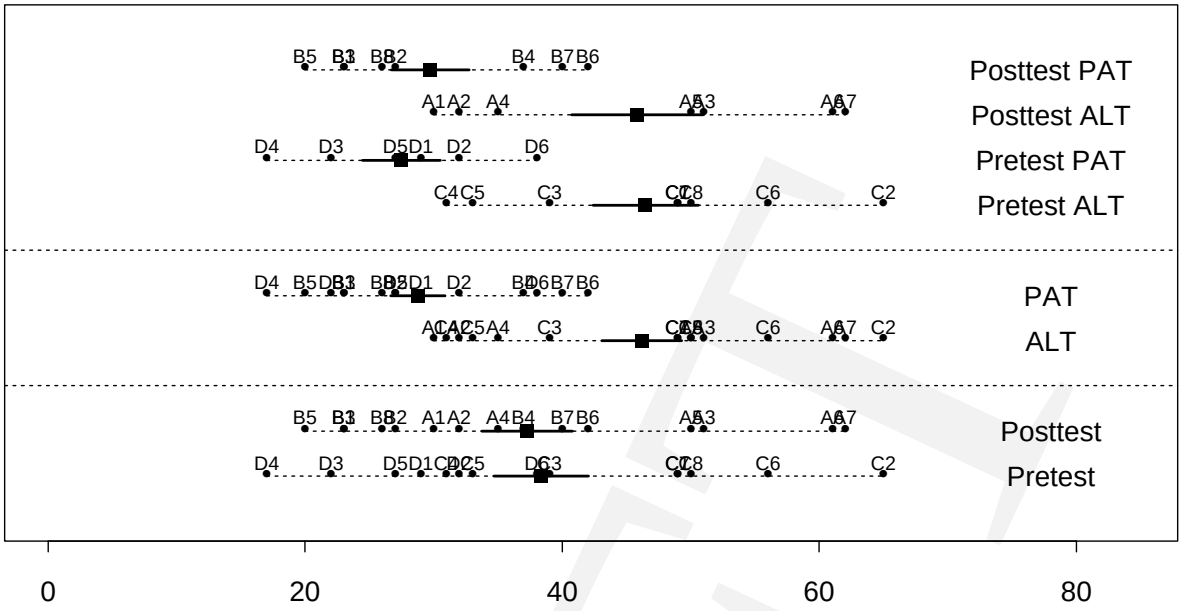

Figure 4: Time required for program *Communication Channels* work task 1.

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.

> **Expectation CO4$_P$:** The ALT group will be faster than the PAT group.
>
> **Expectation CO5$_P$:** The PAT group commits more errors than the ALTgroup.

Results: The results as shown in Figure 5 are inconclusive with respect to CO4$_P$, because the ALT group is much faster in the pretest than in the posttest. This is unexpected except if we assume the subjects could not concentrate well enough in the (afternoon) posttest. However, this assumption is not well supported by the correctness of the solutions, which was just as high in the posttest (8 out of 15) as in the pretest (8 out of 14). Although overall (pretest plus posttest) ALT is indeed faster than PAT ($p = 0.086$), the difference stems mainly from the pretest ($p = 0.000$), whereas ALT is slower than PAT in the posttest ($p = 0.409$). Therefore, we can not clearly decide CO4$_P$.

CO5$_P$ is also not clearly supported. The error rate in the ALT groups is almost as high as in the PAT groups.

4.3.3 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 right order. (A similar sequence problem plagued the ALT subjects in task 1.)

Expectations: Regarding the work task above, we have the following expectations:

> **Expectation CO6$_P$:** The time required for the PAT group is higher than the time required for the ALT group.
>
> **Expectation CO7$_P$:** The PAT group commits more errors than the ALTgroup.

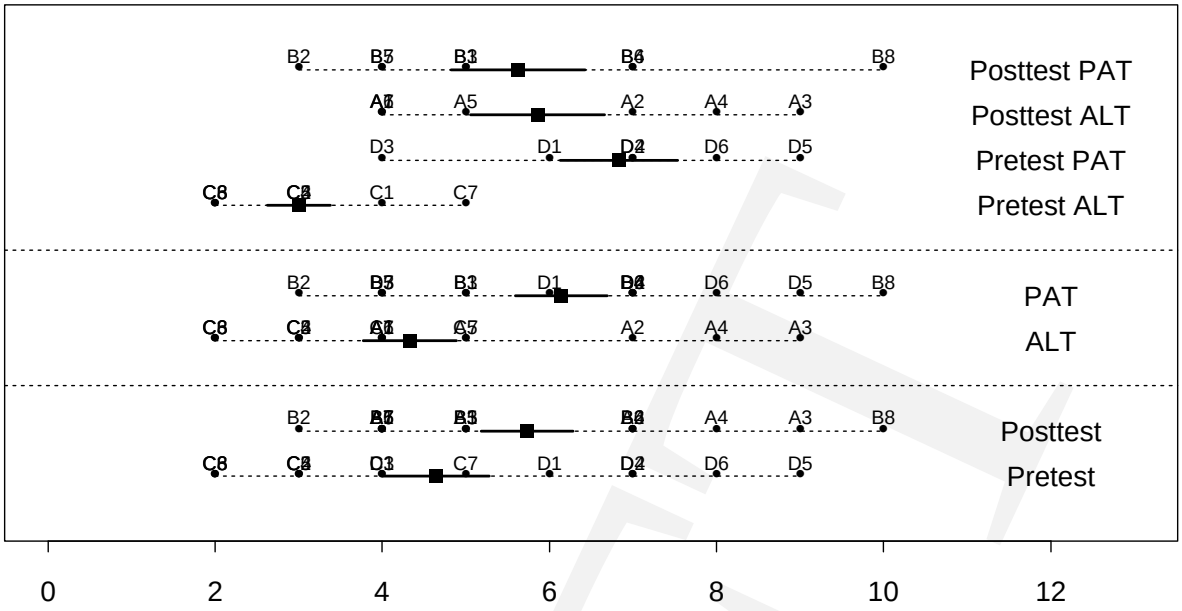

Figure 5: Time required for program *Communication Channels* work task 2.

Results: $CO6_P$ is supported (see Figure 6): Overall, the ALT group is significantly faster ($p = 0.000$) than the PAT group.

Expectation $CO7_P$ is also supported. While in the ALT groups no errors were observed, we counted 6 wrong solutions (out of 14) for PAT. Comparing these results to those of $CO2_P$ we find that for the CO program the ALT problems during program extension are compensated by PAT problems during object creation (which is more frequent in practical programming). However, the object creation problem could be overcome by a suitable convenience method. Summing up all three tasks we conclude that the pattern use in this program was entirely beneficial.

4.4 Composite and Abstract Factory: Graphics Library (GR)

Program description: *Graphics Library* (or GR, for short) 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, the generator, the output device is selected. Depending on the output device the corresponding types of graphical objects are created. Some basic 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, which can be manipulated like individual objects.

Patterns used in the PAT version of this program are ABSTRACT FACTORY and COMPOSITE. The ABSTRACT FACTORY creates the corresponding products depending on the selected graphical output device by instantiating the desired concrete factory. The concrete factory instantiates only those classes that the selected graphical device can handle. The COMPOSITE combines graphical objects (including groups) into groups hierarchically and manipulates groups transparently.

The ALT version of the program realizes the instantiation of the appropriate classes for each graphical output device by switch-statements in 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 special and 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

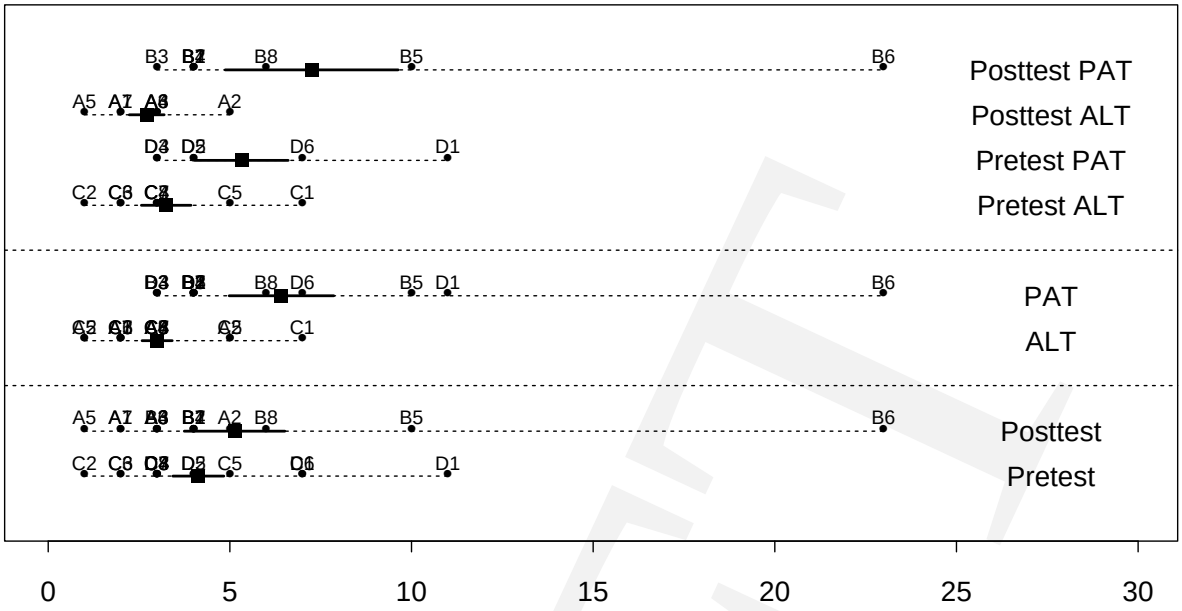

Figure 6: Time required for program *Communication Channels* work task 3.

B individually to A.

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

4.4.1 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 all methods of the generator class in which a switch-statement decides what product is instantiated. 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 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 ALT program to be slightly easier to understand.

Pattern knowledge will help both groups 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.

> **Expectation GR1$_P$:** Subjects working on the ALT program version are slightly faster than subjects working on the PAT version.
>
> **Expectation GR2$_k$:** For both program versions the posttest groups will be faster than the pretest groups.

Results: The results shown in Figure 7 support GR1$_P$. Both groups maintaining the ALT program were faster than the corresponding PAT groups with the same pattern knowledge level (together: significance $p = 0.10$).

Expectation GR2$_k$ is also supported. The improvement from the pretest to the posttest is 17.3% ($p = 0.17$) for the PAT group and 22.8% ($p = 0.031$) for the ALT group. That is 21.2% overall ($p = 0.021$).

4.4.2 Work task 2

This work task was a simple understanding test concerning the COMPOSITE structure. The

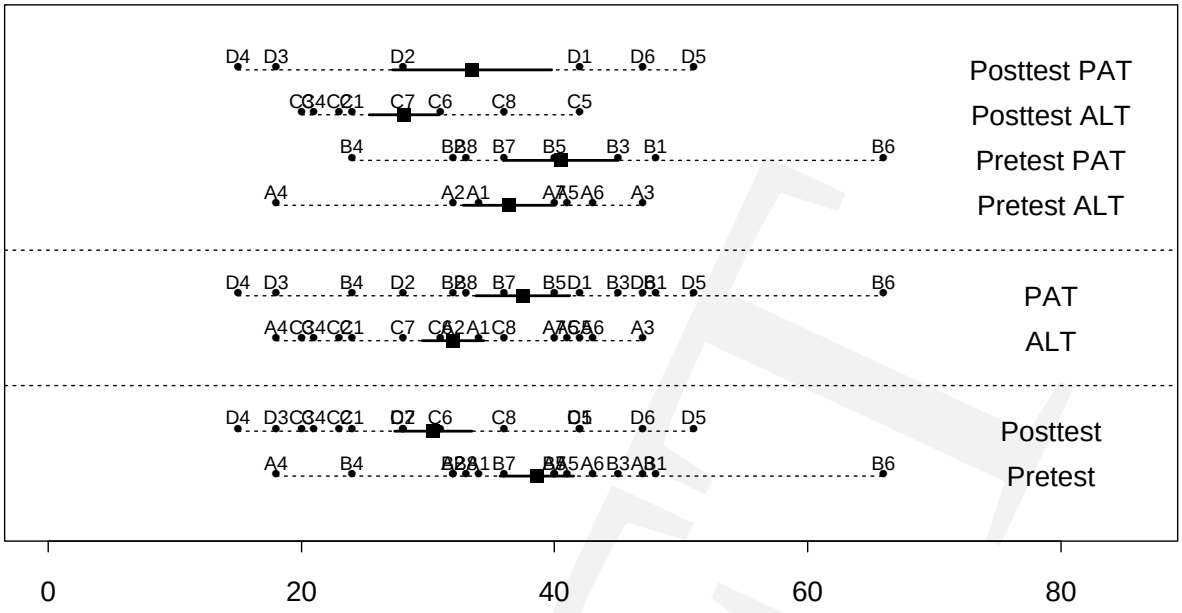

Figure 7: Time required for program *Graphics Library* work task 1.

question was whether or not a certain sequence of operations would result in an x-shaped figure. Subjects of both groups had to examine the COMPOSITE structure; the key to the answer was 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 at the spot of interest. So we do not expect to observe significant differences between the ALT and the PAT groups. But we expect a difference between pretest and posttest: subjects without pattern knowledge are slower than subjects with pattern knowledge because subjects with pattern knowledge are familiar with the structure of the COMPOSITE.

> **Expectation GR3$_P$:** There is no difference between the PAT and ALT groups.
>
> **Expectation GR4$_k$:** Subjects in the posttest are faster than subjects in the pretest.

Results: As we see in Figure 8, the difference between PAT and ALT ($p = 0.085$) is very similar to the difference between pretest and posttest ($p = 0.091$). Both are only weakly significant. In particular, both depend on whether the large value of one subject in the ALT pretest group is an outlier or not. Given the structure of the programs and comparing the individual results within the pretest PAT and ALT groups, we tend to consider it an outlier. In this case GR3$_P$ would be supported and GR4$_k$ would not, meaning that the COMPOSITE (compared to the given alternative solution) and the ABSTRACT FACTORY are just as good as their alternatives in program GR and do not depend on pattern knowledge.

5 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 which over-kill the problem at hand is useful, harmful, or neutral.

We found that all of this can be the case, depending on the situation. Software engineering common sense turned out to be a pretty accurate predictor of these effects for three out of the four programs considered in this exper-

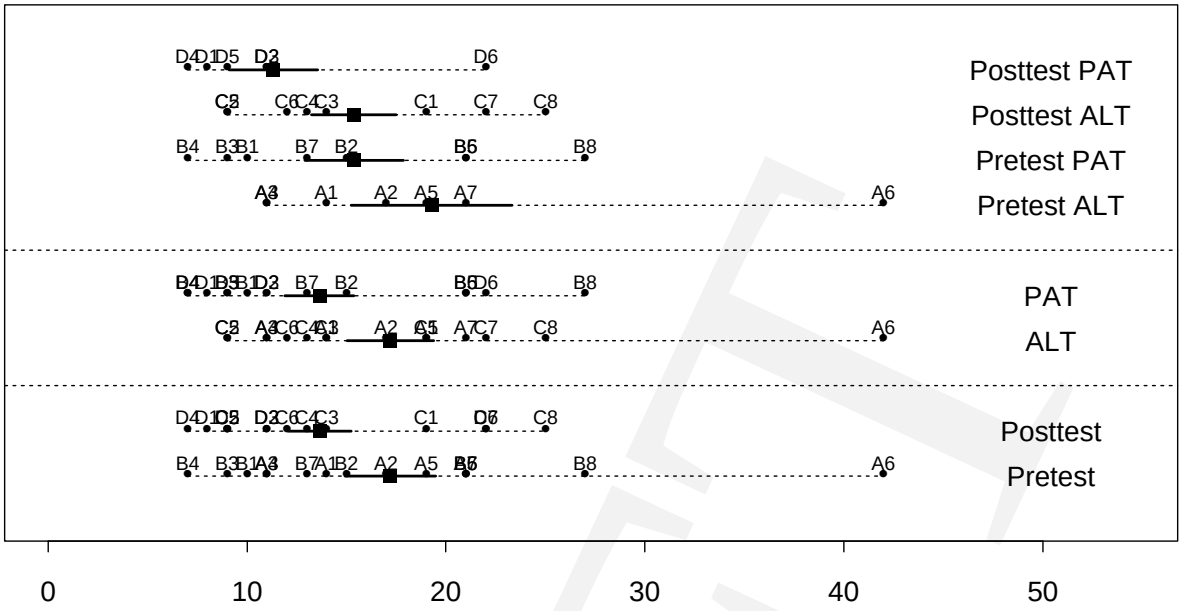

Figure 8: Time required for program *Graphics Library* work task 2.

iment. Summarizing the individual expectations versus actual observed 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 pattern, in particular for subjects with low pattern knowledge.

- Program "Boolean Formulas (BO)" (Composite, Visitor):
  Expectation: The Visitor is difficult to understand and thus harmful.
  Actual result: The Visitor is neutral.

- 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: Just that.

- Program "Graphics Library (GR)" (Composite, Abstract Factory):
  Expectation: The two versions are quite similar and should not make much of a difference.
  Actual result: They did not make much of a difference.

We suggest the following four lessons learned. First, it is not always useful to use a design pattern if there are simpler alternatives, but, second, patterns are sometimes superior to the simpler solutions in non-obvious ways. Third, use software engineering common sense to decide which design solution to prefer. Fourth, a thorough understanding of design patterns often helps when maintaining programs using them, even if these programs are neither very large nor very complicated.

We add, however, 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.

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 tradeoffs involved similar to the ones discussed here? What are the effects of pattern versus non-pattern designs for *long term* maintenance in-

volving many interacting changes? How does the use or non-use of patterns influence activities other than pure maintenance, e.g. inspections or code reuse? Can we characterize the situations in which our common sense will mislead us? Or, put the other way round: What facts and rules need to be added to our common sense to make it more accurate for design decisions?

Acknowledgements

We thank Ernst Denert for making the experiment possible and all our subjects for being so interested in it.

References

[1] K. Beck, J.O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlissides. Industrial experience with design patterns. In *18th Intl. Conf. on Software Engineering*, pages 103–114, Berlin, March 1996. IEEE CS press.

[2] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):., . 1996.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley and Sons, Chichester, UK, 1996.

[4] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit, editor, *11th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, pages 472–495, Jyväskylä, Finland, June 1997. Springer Verlag.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[6] Lutz 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.

[7] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F. Tichy. Two controlled experiments assessing the usefulness of design pattern information during program maintenance. *Empirical Software Engineering*, .(.):., . 1998. Submitted. http://wwwipd.ira.uka.de/~prechelt/Biblio/.

[8] Lutz Prechelt, Barbara Unger, and Douglas Schmidt. Replication of the first controlled experiment on the usefulness of design patterns: Detailed description and evaluation.

Technical Report wucs-97-34, Washington University, Dept. of CS, St. Louis, December 1997.