Submission (December 18, 1997) to 'Empirical Software Engineering'

Two Controlled Experiments Assessing the Usefulness of Design Pattern Information During Program Maintenance

Lutz Prechelt, Barbara Unger, Michael Philippsen, Walter Tichy
Universität Karlsruhe, Fakultät für Informatik

(Received ..... ; Accepted in final form .....)

Abstract. This paper reports on two controlled and repeatable experiments investigating whether software design patterns improve software quality and programmer productivity during software maintenance.

Subjects performed maintenance tasks on two programs ranging from 360 to 560 LOC including comments. Both programs contained design patterns. The controlled variable was whether the design patterns were documented or not. The experiments thus tested whether pattern documentation helps during maintenance, provided patterns are present.

The experiment was initially performed with 74 graduate students at the University of Karlsruhe, Germany, with programs written in Java. The experiment was repeated with 22 undergraduate students at Washington University in St. Louis, USA, with the programs rewritten in C++.

A conservative analysis of the results supports the hypothesis that pattern-relevant maintenance tasks are completed faster and with fewer errors if pattern documentation is provided. The results also suggest that the positive effects of pattern documentation do not rely on a particular programming language and background.

Key words: controlled experiment, design pattern, documentation, maintenance

1. Introduction

One of the most difficult tasks in software engineering is finding a good design and then working according to it. Software design patterns are thought to help in this situation. A software design pattern describes a proven solution to a software design problem with the goal of making the solution reusable. One might say that design patterns are to programming-in-the-large what algorithms are to programming-in-the-small: Both provide proven solutions to known problems, encouraging reuse and relieving programmers of reinvention.

The idea of design patterns has quickly caught the attention of practitioners and researchers, and the pattern literature is burgeoning. The first systematic collection of design patterns was published by Gamma, Helms, Johnson, and Vlissides (GHJV95) (nicknamed the "Gang of Four Book"). Shortly thereafter, additional patterns were reported by

Buschmann et al. (BMR$^+$96). The book by Garlan and Shaw (SG96) also provides a wealth of patterns for software architecture. Annual workshops are being held (Sch97) to promote pattern mining and a consistent style of reporting patterns. Pattern papers show up in other software conferences as well, reporting on new patterns, pattern taxonomies, and pattern tools. Formalizations of patterns are sought and tools are being built for pattern mining, matching known patterns in existing software, and programming with patterns.

The main advantages claimed for design patterns, according to the pattern literature, are as follows:

1. Using patterns improves programmer productivity and program quality;

2. Novices can greatly increase their design skills by studying and applying design patterns;

3. Patterns encourage best practices even for experienced designers;

4. Design patterns improve communication, both among developers and from developers to maintainers.

1.1. Our experiments

The experiments reported here represent the first attempts at testing in a repeatable and controlled manner claim 1 above.[1] The experiments are set in a maintenance context. Assume a maintainer knows what design patterns are and how they are used. Furthermore, assume that a program was designed and implemented using patterns. Now the question is:

> Does it help the maintainer if the design patterns in the program code are documented *explicitly*, as opposed to a documented program structure without reference to design patterns.

We investigate this question in the following manner: Several subjects receive the same program source code and the same change requests for that program; they have to provide appropriate changes sketched on paper (first experiment) or as operational program code (second experiment). The change requests concern those aspects of the program that are implemented using design patterns. The program is documented in detail but the subjects in the control group receive no explicit information about design patterns in the program, whereas the experiment group receive the program with the design patterns explicitly marked and named in a small number of additional comments (called *pattern documentation* or PD). Subjects are assigned randomly to the groups. We investigate whether and how the performance of the two groups differs by measuring completion time, grading answers, and counting correct solutions.

The experiments were performed with a total of 96 student subjects, each working in a single session of 2 to 4 hours. The tasks were based on two programs about 6 to 10 printed pages in length.

1.2. Related work

Design patterns are a recent idea, so it is not surprising that evidence about their effectiveness is scarce. Case study reports and anecdotal evidence of positive effects can be found in (BCC$^+$96; GHJV95). Part of the program maintenance literature is loosely relevant to pattern effectiveness, but we have found no reports that specifically address design patterns as an aid to maintenance. Likewise, as far as we know, the design pattern community itself has not yet undertaken controlled experiments to test design pattern claims.

1.3. Structure of the article

The next section will describe the design and implementation of the experiments including a statement of the hypotheses, a description of the subjects' background, a description of the tasks, and a discussion of possible threats to the internal and external validity of the experiments. Section 3 discusses the results and Section 4 summarizes the results and raises questions for future research.

Due to space restrictions, we cannot provide a complete description of the programs and tasks used. However, detailed information is available in two technical reports (Pre97; PUS97) that include the original experiment materials such as the task descriptions and source program listings.

2. Description of the experiments

The first experiment was performed in January 1997 at the University of Karlsruhe (Uka), the second in May 1997 at Washington University St. Louis (Wustl). Although the experiments were similar, there were some variations. We will therefore describe the experiments separately and refer to them as Uka and Wustl, respectively. Where appropriate, we will give information for Uka first and append the corresponding information for Wustl in angle parentheses ⟨like this⟩.

2.1. Hypotheses

First we need to define the concept of pattern-relevance. A maintenance task on a program is *pattern-relevant* if (1) the program contains one or more software design patterns and (2) a grasp of the patterns in the program is expected to simplify the maintenance task.

The experiments aimed at testing the following hypotheses:

Hypothesis H1: With PD, pattern-relevant maintenance tasks are completed faster than without.

Hypothesis H2: With PD, fewer errors are committed in pattern-relevant maintenance tasks than without.

Speed of task completion is measured in time (minutes). The number of errors are quantified by assigning points and by counting correct solutions.

2.2. Subjects and environment

The 74 ⟨22⟩ subjects of the Uka ⟨Wustl⟩ experiment were 64 ⟨0⟩ graduate and 10 ⟨22⟩ undergraduate computer science students.

They had taken a 6-week ⟨12-week⟩ intensive ⟨standard⟩ lecture and lab course on Java ⟨C++⟩ and design patterns before the experiment. On average, their previous programming experience was 7.5 years ⟨5 years⟩ using 4.6 ⟨4.0⟩ different languages with a largest program of 3510 LOC ⟨2557 LOC⟩. Before the course, 69% ⟨76%⟩ of the subjects had previous experience with object-oriented programming, 58% ⟨50%⟩ with programming GUIs.

The subjects had sufficient theoretical knowledge of design patterns, as indicated by a pattern knowledge test conducted at the start of each experiment. For those patterns that were relevant in the experiment, the Uka subjects' pattern knowledge was better than that of the Wustl subjects, because the Uka course had directly been targeted at the experiment but the Wustl course had not. For some of the relevant patterns, the Wustl subjects had no *practical* experience with these patterns, in contrast to the Uka subjects.

Each of the experiments was performed in a single session of 2 to 4 hours. The Uka subjects had to write their solutions on paper. The Wustl subjects implemented their solutions on Unix workstations.

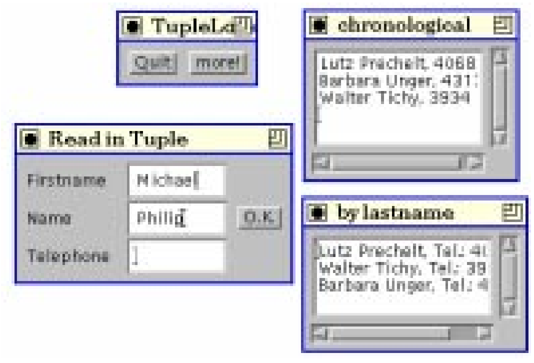
2.3. Programs used

Each subject worked on two different programs. Both programs were written in Java ⟨C++⟩ using design patterns and were thoroughly commented.

Program *And/Or-tree* is a library for handling And/Or-trees of strings and a simple application of it. It has 362 ⟨498⟩ LOC in 7 ⟨6⟩ classes; 133 ⟨178⟩ of these LOC contain only comments, and an additional 18 ⟨22⟩ lines of PD were added in the version with PD. *And/Or-tree* uses the Composite and the Visitor design pattern (GHJV95).

Program *Phonebook* is a GUI program for reading tuples (name, first name, phone number) entered by the user and showing them in different views on the screen, see the screenshot in Figure 1. Because the WUSTL subjects had not learned a GUI library in the course, the C++ version of *Phonebook* is stream-I/O-based: it reads all of its inputs from the keyboard and completely redisplays all views to standard output after each change. *Phonebook* has 565 ⟨448⟩ LOC in 11 ⟨6⟩ classes; 197 ⟨145⟩ of these LOC contain only comments, and an additional 14 ⟨10⟩ lines of PD were added in the version with PD. *Phonebook* uses the Observer and the Template Method design pattern (GHJV95).

See (Pre97; PUS97) for the full source code of the programs.



Firstname: Michael
Name: Philig

Lutz Prechelt, 4068
Barbara Unger, 431
Walter Tichy, 3934

Lutz Prechelt, Tel.: 4
Walter Tichy, Tel.: 39
Barbara Unger, Tel.: 4

Figure 1. Screenshot of UKA *Phonebook* program

2.4. EXPERIMENT CONTROLS, GROUP SIZES

The independent variable in both experiments was the presence or absence of design pattern documentation (PD) as comments in the source programs.

We used a counterbalanced experiment design (Chr94), see Table I: The first variable is the order in which a subject receives the two programs. One of those programs was supplied with PD, the other without. This design results in a second variable, i.e., the order in which PD is received: first with, then without PD, and vice versa. The combination of the variables results in four groups. The subjects did not know in

Table I. The four experiment groups and their size. The number of data points is one per subject, except for those subjects that did not complete the respective task, but dropped out of the experiment instead. For UKA there was no such mortality. See also Section 2.7. (A^+P^- stands for "first perform *And/Or-tree* with PD, then perform *Phonebook* without PD" and so on.)

	first with PD then w/o PD	first w/o PD then with PD
groups: first *And/Or-tree*, then *Phonebook*	A^+P^-	A^-P^+
—UKA initial no. of subjects	19	18
—UKA no. of data points, both tasks	19	18
—WUSTL initial no. of subjects	6	5
—WUSTL no. of data points, *Phonebook*	4	3
—WUSTL no. of data points, *And/Or-tree*	4	4
groups: first *Phonebook*, then *And/Or-tree*	P^+A^-	P^-A^+
—UKA initial no. of subjects	18	19
—UKA no. of data points, both tasks	18	19
—WUSTL initial no. of subjects	6	5
—WUSTL no. of data points, *Phonebook*	3	3
—WUSTL no. of data points, *And/Or-tree*	4	4

advance whether a program would contain PD or not; they did not even know that PD would be a treatment variable.

2.5. TASKS

For *And/Or-tree*, each subject received the following 4 subtasks: (1) Find the right spot for a particular output format change, (2) give an expression to compute the number of variants represented by a tree, (3) create an additional visitor class that computes the number of variants faster (similar to an already existing class computing depth information), and (4) instantiate such a visitor and print its result. Subtasks (3) and (4) are considered pattern-relevant.

For *Phonebook*, each UKA subject received the following 5 subtasks: (1,2) Find two spots for small program changes (output format change, window size change), (3) create an additional observer class using a Template Method,[2] (4) instantiate and register such an observer, (5) create an additional observer class similar to an already existing one not using a Template Method. Subtasks (3) to (5) are considered pattern-relevant.

There are two important differences between UKA and WUSTL regarding *Phonebook*. First, in UKA subtask (3) a similar class was already present in the program. Subtask (3) could thus be solved by imitation; this was not true for WUSTL. Second, subtasks (2) and (5) were not required in WUSTL and (4) was implicit in (3).

For the class creation subtasks, only the interface of the class needed to be written; the actual implementation was not required, although the WUSTL participants were asked to provide a complete solution if they easily could.

2.6. Measurements

For each task (but not for each subtask) of each subject we measured the time between handing out and collecting the experiment materials. It is unclear how the time spent for general program understanding could be distributed among the subtasks, so no subtask time information was collected. For each subtask, we graded the answers according to the degree of requirements fulfillment they provided. The grades were expressed in points. There was a total of 2+2+8+3=15 points ⟨2+2+8=12⟩ for the UKA ⟨WUSTL⟩ subtasks of *And/Or-tree* and 2+3+8+4+6=23 points ⟨2+8+8=18 points⟩ for those of *Phonebook*. Since graded point scales are somewhat subjective we also recorded the number of completely correct solutions.

2.7. Threats to internal validity

The main independent variable in these experiments is the presence or absence of pattern documentation (PD). Unavoidably, though, adding or removing PD also changes the amount of overall documentation in the program, because there is no appropriate placebo that could be used in its place. Hence, it is impossible to distinguish between the effects of adding PD (as such) and the effects of adding some documentation (of whatever kind). One can think of two possibilities for the latter: First, adding documentation may improve performance, because the program is described better. Second, adding documentation may also hamper performance, because more documentation takes more time to digest and may increase the cognitive load or introduce additional stress.

No matter which effect is dominant, it has not influenced our results much, because our programs were thoroughly documented even without PD and the amount of additional documentation was quite small (between 2% and 5% of all source lines). Therefore we can expect our results to show effects from adding PD, not effects from adding *any* documentation.

Apart from the above, all relevant external variables have been appropriately controlled in this experiment; furthermore, the counter-balanced experiment design would even (partially) compensate for unbalanced subject ability between the groups, should it have occurred by chance.

The dominant control problem is mortality: Some WUSTL students gave up on a task when they thought it would be too difficult for them or take too long. Four students gave up on both tasks. Fortunately, mortality occurred almost exactly as often in the groups with PD as in those without PD. By ignoring incomplete tasks entirely, it is therefore safe to assume that the mortality does not bias the results. See Table I for the resulting group sizes; there was no mortality in UKA.

We applied manual and automated consistency checks for guarding against mistakes in data gathering and processing.

2.8. THREATS TO EXTERNAL VALIDITY

There are several sources of differences between the experimental and real software maintenance situations that limit the generalizability (external validity) of the experiments: in real situations there are subjects with more experience, often working in teams, and there are programs and change tasks of different size or structure.

Experience: The most frequent concern with experiments using student subjects is that the results cannot be generalized to professionals, because the latter are more experienced. In the present case, professional programmers may have less need for PD because of their experience. But just as well they may be able to exploit it more profitably than our student subjects.

Team work: Realistic programs are always team work. Individual change tasks during maintenance may also often be performed by more than one programmer. Such cooperation requires additional communication about the program. In this case PD may have further advantages, not visible in the experiment, because one of the major (purported) advantages of design patterns is to provide adequate common terminology.

Program size and structure: Compared to typical industrial size programs, the experiment programs are rather small and simple. This property does not necessarily invalidate the results of the experiments, though. If a positive effect is found, it is plausible that increasing program complexity magnifies the effect, because PD provides program slicing information. For pattern-relevant tasks, PD points out which parts of a program are relevant and enables one to ignore the rest; such information becomes more useful as more code can be ignored.

Change tasks and pattern relevance: The benefits from PD may be smaller in reality than in our experiment for two reasons, but the exact effects are quite unclear and may depend on the domain. First, if the programmer must understand a large number of design patterns, his/her understanding of each individual pattern may be reduced or confused and therefore less helpful. Second, for many change tasks, design patterns may not be relevant at all.[3]

Only repetition of similar experiments with professionals on real programs and real maintenance tasks can evaluate these threats. On the other hand, the experiment was biased in several ways towards showing smaller-than-reality benefits from PD; see the discussion in the conclusion. Note in particular that knowing and using more design patterns overall increases the effective pattern density in a program and increases the amount of information that PD can provide.

3. Results and discussion

The most salient results of both experiments are summarized in Tables II and III. A part of the results is also visualized in Figure 2. For WUSTL, the results of subjects that did not deliver a particular task were ignored for that task. For UKA, all subjects delivered both tasks.

3.1. RESULTS FOR *And/Or-tree*

From Table II the UKA results for *And/Or-tree* at first appear to be inconclusive: While the A^+ group (i.e., the group with PD) obtained slightly more points on average (lines 1 and 2), it consumed significantly more time (line 4). However, this observation is misleading, because the non-computerized working environment made it difficult for a subject to check whether a solution was correct. In real software maintenance, incorrect solutions would be detected and corrected, taking additional time not observed in the experiment. In the UKA experiment, incorrect 'solutions' are produced quickly.

It turns out that most such quick but incorrect solutions occur in the A^- group: Subtask (3) was solved correctly by 15 subjects of the A^+ group with PD, but only by 7 of the A^- group without PD (line 3). This difference is significant ($\chi^2 = 3.55$, $p = 0.060$, Fisher exact $p = 0.051$). If we consider only the correct solutions the time difference vanishes

Table II. Results for the *And/Or-tree* task. Columns are (from left to right): line number, name of variable, arithmetic average PD^+ of sample of subjects provided with design pattern information (PD), ditto without, 90% confidence interval I for difference $A^+ - A^-$ (measured in percent of A^-) or $P^+ - P^-$, significance p of the difference (one-sided). I and p were computed using Bootstrap resampling (ET93) with 10000 trials because many distributions were distinctly non-normal. "Relevant points" are the points for the pattern-relevant subtasks only.

	Variable	mean with PD	mean w/o PD	means difference (90% confid.) I	significance p
UKA, program *And/Or-tree*:					
1	all points	11.1	10.4	−8.2% ... + 22%	0.23
2	relevant points	8.5	7.8	−7.7% ... + 23%	0.20
3	#corr. solutions	15 of 38	7 of 36		
4	time (minutes)	58.0	52.2	−3.0% ... + 24%	**0.094**
5	— corr.	52.3	45.4	−11% ... + 41%	0.17
6	— best 7	38.6	45.4	−37% ... + 6.3%	0.13
WUSTL, program *And/Or-tree*:					
7	all points	9.8	10.0	−18% ... + 13%	0.48
8	relevant points	6.7	6.5	−12% ... + 19%	0.28
9	#corr. solutions	4 of 8	3 of 8		
10	time (minutes)	52.1	67.5	−43% ... − 0.5%	**0.046**

(line 5). Moreover, to remove bias we must compare only the fastest 7 subjects of each group with correct solutions. In this case the time difference reverses and the A^+ group is faster (line 6). See also Figure 2.

Thus, it is safe to say that although the advantage of having PD is blurred due to the conditions of the experiments, it is still visible: Far more subjects with PD were able to come up with completely correct solutions than without and also tend to be faster then.

The WUSTL results for *And/Or-tree* are even clearer: Here we find essentially no difference in the number of points (lines 7 and 8) or the number of completely correct solutions (line 9), but a large advantage in the time required for the group with PD (line 10). With a confidence of 0.9, PD saved between zero and 43 percent of the maintenance time for this task.

As for learning effect, the UKA subjects were on average significantly faster (but did not obtain more points) in their second task. The acceleration hardly interacts with the presence of PD. The learning effect is compensated by the counterbalanced design and is not relevant for the interpretation of the results. The learning effect could not be assessed for WUSTL, because the group sizes were too small.

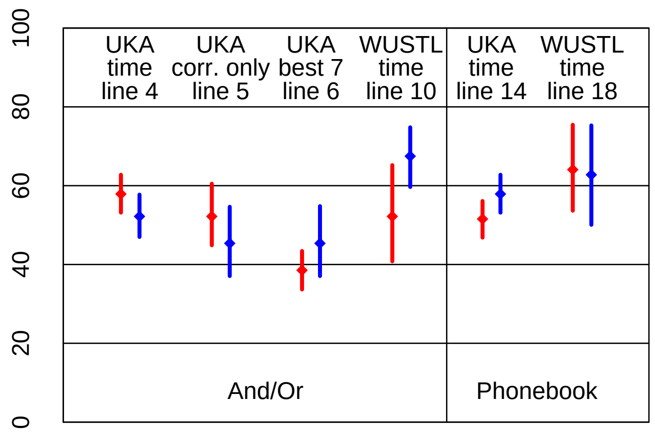


Figure 2. Graphical display of time entries (in minutes) from Tables II and III: The left plot of each pair represents the group with PD, the right one the group without PD. The dot marks the mean of the task completion time, the strip indicates a 90% confidence interval for the mean. "line n" indicates the corresponding line in Table II or III.

Table III. Results for the *Phonebook* task.

	Variable	mean with PD A^+ or P^+	mean w/o PD A^- or P^-	means difference (90% confid.) I	significance p
Uka, program *Phonebook*:					
11	all points	20.8	21.1	$-6.0\% \ldots +3.3\%$	0.35
12	relevant points	16.1	16.3	$-8.0\% \ldots +4.0\%$	0.35
13	#corr. solutions	17 of 36	15 of 38		
14	time (minutes)	51.5	57.9	$-22\% \ldots +0.3\%$	**0.055**
Wustl, program *Phonebook*:					
15	all points	12.3	14.2	$-33\% \ldots 5.6\%$	0.12
16	relevant points	10.5	12.2	$-38\% \ldots 8.5\%$	0.15
17	#corr. solutions	1 of 6	1 of 7		
18	time (minutes)	64.1	62.7	$-23\% \ldots 29\%$	0.45

3.2. Results for *Phonebook*

For *Phonebook*, the results (as shown in Table III and Figure 2) are clear for Uka, but blurred by experiment artifacts for Wustl.

The *Phonebook* results of Uka show essentially no difference in the total number of points per subject (line 11), the number of points for the pattern-relevant subtasks 3 to 5 (line 12), or the number of solutions that were completely correct for pattern-relevant subtasks (line 13). The rather high average point values obtained indicate that the task was simple for these subjects.

Still, however, the group with PD managed to solve the task significantly faster than the group without PD (line 14). The advantage can also be quantified: it has an expected size somewhere between zero and 22 percent with confidence 0.9. See also Figure 2.

The WUSTL results for *Phonebook* are completely inconclusive: Quantitatively, the times for the two groups are essentially the same (line 18) and the PD group obtained somewhat fewer points (lines 15 and 16). Qualitatively, however, it is clear that these results are not meaningful, because there is only a single correct solution (line 17)! Our interpretation is that the task was just too difficult for these subjects for two reasons. First, the non-GUI presentation style of the WUSTL *Phonebook* program made the use of the Observer pattern rather unintuitive and obscure. Second, these subjects had never actually implemented an Observer and there was no example class they could imitate (as the UKA group could). Our results suggest that under such circumstances, PD might be worthless. Obviously, the results for the UKA and WUSTL variants of the *Phonebook* task must not be compared directly.

As for learning effect, the same discussion applies as for *And/Or-tree* above; see (Pre97) for details.

4. Interpretation and conclusions

We will argue now why the results from our experiments suggest positive effects of PD in pattern-relevant maintenance tasks.

The design of these experiments was extremely conservative; many design decisions biased them towards *not* showing any effects from adding PD:

1. The subjects knew they would participate in an experiment "about design patterns", so they were keyed to look for patterns in the programs. Such a context may reduce the benefits from PD.
2. In software production reality, a software engineer might know a lot more different patterns, some of them quite similar to each other, so that any single PD would transport more information.
3. The programs were rather small, so even without PD the subjects could achieve good program understanding within a reasonable time. Again, in reality PD might be more helpful if the fraction of program understanding effort that PD can save grows with the size of the program.

4. The programs were thoroughly commented, not only at the statement level, but also at the method, class, and program levels. Thus, the subjects had sufficient documentation available for program understanding even without PD. In contrast, most programs in the real world lack design documentation. PD might be a good means to improve design documentation, as it is rather compact.

Given these circumstances, we expect performance advantages through PD to be more pronounced in real situations than in our experiments.

In summary we find that our results support both of our hypotheses introduced in Section 2.1:

Hypothesis H1 (Pattern-relevant maintenance tasks will be completed faster with PD): This hypothesis is clearly supported by the UKA *Phonebook* results and the WUSTL *And/Or-tree* results. The other two results are inconclusive; however, the opposite of this hypothesis is not supported at all. Where found, the size of the effect (0 to 40 percent speedup) is considerable, although this is certainly not directly generalizable to other circumstances.

Hypothesis H2 (Fewer errors will be committed in pattern-relevant maintenance tasks with PD): This hypothesis is clearly supported by the UKA *And/Or-tree* results. The other three results are inconclusive; however, the opposite of this hypothesis is not supported at all, judging from the fraction of completely correct solutions, which is the only reliable quality measure.

We conclude that depending on the particular program, change task, and personnel, PD in a program may considerably reduce the time required for a program change and may also help improve the quality of the change. Given that we used subjects from two different continents and two different programming languages, the results are not specific to only one type of culture or educational background or to only one programming language.

We therefore recommend that design patterns always be documented explicitly in program source code.

Further work should perform related experiments in different settings. The following questions appear most important. First, how do the effects change for larger programs? Second, how do they change for more difficult tasks? Third, how do the effects change when much larger numbers of different patterns come into play — often with overlap between their instances? Fourth, how do they change when multiple programmers have to cooperate (and hence communicate) in order to make a change? Fifth, what are the effects if programs are more or less undocumented (or even ill-documented) and how, in general, does PD interact with other documentation? Sixth, is PD also helpful during inspections?

Moreover, empirical studies of existing software should determine what fraction of the change tasks (or change effort) is pattern-relevant. The question of design stability also needs to be addressed: We have found initial evidence that PD may slow down architectural erosion and drift (PW92), i.e., delay the decay of the original software design structure. Finally, and perhaps most interestingly, how does maintenance compare for software with patterns versus equivalent software without?

ACKNOWLEDGEMENTS

We thank Douglas Schmidt for the opportunity to repeat the experiment. We also thank all our experimental subjects for making this exciting project possible.

Notes

[1] The experiments also tests the fourth claim, but only indirectly.

[2] Of course the task description did *not* explicitly mention that it was an observer that should be added, etc.

[3] An important question in this context is the *pattern density* in a program: what fraction of the program is a part of a design pattern instance? Little such information is available. An investigation of the Java Abstract Windowing Toolkit (AWT 1.0) and NEXTstep v2 found that 74% and 66% of all classes, respectively, participated in some design pattern instance (Gra97).

References

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.

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.

Larry B. Christensen. *Experimental Methodology.* Allyn and Bacon, Needham Heights, MA, 6th edition, 1994.

Bradley Efron and Robert Tibshirani. *An introduction to the Bootstrap.* Monographs on statistics and applied probability 57. Chapman and Hall, New York, London, 1993.

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

Oliver Gramberg. Counting the use of software design patterns in Java AWT and NeXTstep. Technical Report 19/1997, Fakultät für Informatik, Universität Karlsruhe, Germany, December 1997. to appear, ftp.ira.uka.de.

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.

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.

Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 1992.

Douglas Schmidt. Collected papers from the PLoP '96 and EuroPLoP '96 conferences. Technical Report wucs-97-07, Washington University, Dept. of CS, St. Louis, February 1997. (Conference "Pattern languages of programs").

Mary Shaw and David Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Address for correspondence:
Fakultät für Informatik, Universität Karlsruhe
D-76128 Karlsruhe, Germany
Phone: +49/721/608-4068, Fax: +49/721/608-7343
Email: prechelt,unger,phlipp,tichy@ira.uka.de
WWW: http://wwwipd.ira.uka.de/Tichy/