A Series of Controlled Experiments on Design Patterns: Methodology and Results

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

Abstract

Software design patterns are an idea that is intuitively appealing and has found many advocates. However, as scientists we must be concerned about gathering hard evidence for the claims of beneficial consequences of design patterns. This article describes the major claims and derives the corresponding research questions. It discusses the methodology of a research programme for investigating these questions and sketches the practical constraints that make this research difficult.

It then shortly summarizes three controlled experiments that were successfully carried out within these constraints and lists the main results and their consequences, such as: One should document design patterns when they are used and one must not apply design patterns without judgement of alternatives. Finally, design considerations of a fourth experiment are discussed.

The contribution of this paper is a description of important methodological aspects of practical experimental work and how these relate to the results obtained. Understanding these relations will be important in future empirical software engineering research.

1 The design pattern phenomenon

Software design patterns are basically a methodological engineering idea: Describe a proven solution to a recurring design problem in a standard form chosen such as to clarify the solution's context and properties and to facilitate its reuse in practical situations.

The idea of software design patterns is intuitively appealing and has quickly caught the attention of both practitioners and researchers. The pattern literature is burgeoning. The first systematic collection of design patterns was published by Gamma, Helms, Johnson, and Vlissides [5] (nicknamed the "Gang of Four Book") and was a huge success. Other books followed, e.g. [2, 12], annual workshops are being held [11] to promote pattern mining and a consistent style of reporting patterns. Pattern practitioners published anecdotal reports of successful use of design patterns in real projects and claim various advantages compared to former practice [1].

1.1 The 'object-orientation effect' again?

The enthusiasm with which design patterns are currently being adopted reminds one of the days when object-orientation (o-o) became popular: Huge successes were reported and many believed o-o to be the silver bullet. Only slowly it became clear that there were also problems with properly applying the new capabilities. For a long time there were at least as many projects doing worse (with o-o than without) than there were projects doing better and only slowly the DOs and DON'Ts of o-o were widely understood — a process which is still not complete.

In Computer Science, o-o was quickly adopted as a basis for languages, methods, and tools but hardly any scientist actively investigated the questions *"Is o-o actually better than ordinary structured design and programming? Under which circumstances?"*. Presumably, heavy empirical research on the effects of o-o would have provided a much faster path to understanding and applying o-o properly, but hardly any such work was done. We should not make a similar mistake again.

1.2 Consequences

As we see, software engineering research should actively validate design patterns empirically. Thereby we can not only quantify the advantages, better understand applicability conditions, and thus improve practical pattern application. We will eventually also better understand the mechanisms by which design patterns improve software practice — which should lead to improvements or generalizations of the design pattern idea itself.

In the following section we will discuss a research programme for design pattern validation by first describing claims made and corresponding research question and then discussing applicable research methodology.

Section 3 will then describe an ongoing series of controlled experiments that we conduct in this realm and will present its results so far.

2 Deriving a research programme

We will now sketch a research programme for investigating the properties of the 'design pattern' method idea. First, we list the claims made about possible advantages of design patterns. Each claim X directly leads to two research questions: Number one is "*Is X true?*". Investigating this question will induce further hypotheses which can then be tested in order to answer number two: "*Under which circumstances does X hold?*".

Second, we will discuss the methods by which these questions might be answered and discuss the constraints that make the investigation difficult. Most of this discussion will apply to the empirical investigation of other software engineering topics besides design patterns as well.

2.1 Claims about design patterns

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

Claim LD: Developers **l**earn better design skills fast by studying design patterns.

Claim P: Using patterns improves designer (and maybe programmer) **p**roductivity.

Claim QN: Novices can greatly improve the uality of their designs by applying design patterns.

Claim QE: Even for **e**xperienced developers, patterns encourage best practices and ensure high design **q**uality.

Claim CD: Design patterns improve **c**ommunication among **d**evelopers (both designers and programmers).

Claim CM: Design patterns improve **c**ommunication from developers to **m**aintainers.

The above phrasings could certainly be disputed as the claims are rarely made as explicitly as above. However, we believe the above list is justified from distilling the pattern literature mentioned in the introduction.

2.2 Methodological considerations and practical constraints

We will now discuss scientific and practical aspects of several key ingredients of empirical studies on design patterns: the principal research approach (controlled experiment versus field study), the kinds of participants (students versus professionals), the type of software work investigated (development/maintenance, alone/team etc.), and the technical conduct of the study.

The discussion leads to certain design decisions for the research approach to assessing design patterns.

2.2.1 Laboratory versus field research

The most fundamental distinction in empirical software engineering work is between studies in a 'clean', controlled laboratory setting (controlled experiments, [3]) versus studies 'in vivo', in real software engineering environments (field studies). The latter come either in the form of live observation or as postmortem data analysis ("software archeology"). Note that the laboratory versus field distinction is fuzzy because field studies may exhibit a semi-controlled environment, e.g. in case studies.

The principal advantage of field research is its realism: by construction, the results obtained are applicable to at least one real software engineering situation. Note that this advantage can be misleading: The lack of control and the impossibility of precisely characterizing the field environment may make it very difficult to judge the generalizability of the study.

The principal advantage of laboratory research is reproducibility: The environmental conditions can be neatly controlled and documented. In particular the extremely influential variable 'individual programmer performance' can be controlled by sufficient internal replication, i.e., by statistically comparing random samples of programmers instead of individual ones. As only reproducible results are scientifically 'hard', controlled experiments are in principle preferable to field research.

The problem with laboratory experiments is ensuring that their results are useful: We do still lack models for judging how laboratory results generalize to professional work settings.[1]

The most fruitful overall research approach is usually to use both, laboratory and field research: The results of controlled experiments produce new approaches or hypotheses to be tried and investigated in the field. Conversely, observations in the field produce new hypotheses to be tested by controlled experiments – which is exactly what is currently happening with design patterns: They were developed by practitioners and imply a set of hypotheses to be tested by experiment. Results from both laboratory and field work will be needed to build and validate quantitative models of consequences of design pattern use.

One final note: Rarely, but sometimes, it is possible to conduct controlled experiments in a live software

[1] Note, however, that this problem also applies to field studies. Their results are valid in one realistic context, namely the one where they were performed, but we do lack models for transfering them to other contexts as well. In fact, if we had such models field study generalization was even more difficult than for laboratory studies, because the field environment is much harder to characterize due to its higher complexity.

engineering environment and so get the best of both worlds. See [6] as an example.

2.2.2 Subjects

At the center of any empirical laboratory study are the designers or programmers. These subjects may either be junior or senior software professionals or students. For maximum generalizability one would like to have subjects from a broad cross-section of professional software engineers, representing all types of processes, software domains, experience levels etc. Obviously, this is impossible.

The alternatives all have their disadvantages: Students are possibly inexperienced, which limits the external validity (generalizability) of an experiment. In contrast, senior software professionals will often be highly adapted and specialized to their usual working environment, software domain, and software process. This will also limit external validity, because it is entirely unclear how to transfer results from one homogeneous group of such professionals to another. Junior professionals are in between, mixing the advantages and disadvantages of both groups to some degree; they will not be very different from students close to graduation.

From a practical point of view, professionals will rarely be available at all, because a controlled experiment usually involves significant amounts of nonproductive work due to the required replication. Most experiments will therefore end up being conducted with student subjects. If they are sufficiently experienced, that is not necessarily a disadvantage: Depending on the given tasks, it may be easier to extrapolate student results to higher experience levels than to transfer results coming from one specific professional background to another.

Independent of the kind of subjects, their number plays a crucial role in experimentation: The more subjects are available, the easier it is to discriminate the individual (inter-personal) variations in performance from those variations that stem from the variable under study. If too few subjects are available, the results will often not be statistically significant. The problem becomes smaller if the individual performance is very homogeneous or if a way is known to predict individual performance from personal data.

Furthermore, most experiments require the subjects to have specific knowledge. For example subjects need design pattern knowledge for most types of design pattern experiment. If the experiment concerns a state-of-the-art technique such as design patterns, subjects will often have to be taught a course on the topic of the experiment first.

2.2.3 Task type

There are several dimensions in which the task performed by the experimental subjects may differ: it may be a design or implementation task, from scratch or in maintenance, may be done alone or by a team, may target programs of different size and from different domains, and may employ different types of design patterns.

From the purely pattern-research point of view, large team tasks would usually be most preferable. However, practically, it is unrealistic to obtain sufficient replication for such tasks; the overall amount of work involved would become too large. For increasing replication to a useful amount we may use smaller tasks and let individuals work alone. However, for small tasks there is a danger that the total design pattern content of the task may become too low. Preferring maintenance (such as program extension) over construction from scratch leads a way out of this dilemma, because it allows to use much larger programs without increasing the task size.

2.2.4 Environment

The most realistic working environment would be established if the experimental subjects use their normal workplace: their own office and desk, their own computer hardware and software configuration, their normal environment noises, bookshelf contents, and colleagues to ask.

But depending on the concrete experimental question, this may be counter-productive and/or impossible to control: Perhaps colleagues must not be asked or certain computer help not be used; external interruptions by the telephone or colleagues disrupt the work and influence the results in an uncontrollable and irreproducible manner, etc.

Hence we will usually want to use a clean, well-equipped, quiet laboratory workplace which mimics all essential features of the original environment, but lacks its disadvantages. This is difficult because it is a hard technical challenge to supply dozens of different subjects each with his or her preferred computer setup. The volume of hardware and space required is one problem, the diversity and complexity of the software configurations is another.

Practical constraints will therefore often force the experimenter to either use a standardized computer setup or to conduct the experiment without computer infrastructure, e.g. using handwriting and handdrawing. How harmful these restrictions are depends a lot on the actual task to be solved, thus imposing further constraints on the selection of the tasks. For instance a design task can often be solved on paper alone, while an implementation task usually requires a familiar computing environment. As a rule of thumb, the larger the size of the solution text, the more important a computer environment will be for getting results that generalize to professional working environments. Fortunately, often tasks can be found that result in rather short solution texts.

2.2.5 Summary

As we see from the above discussion, controlled experiments are the method of choice for the initial scientific investigation of design patterns. Such experiments will suggest specific questions for subsequent field research. For the initial experiments, student and professional subjects are both acceptable. Practical considerations suggest that maintenance tasks be investigated first, because the possible program and task size is much larger then, compared to designing and writing a program from scratch. Tasks have to be selected so as to minimize the influence of other technical constraints, in particular the available computing infrastructure.

3 Actual experiments

We have performed three experiments according to the considerations and constraints mentioned above and are currently planning another. All of them will now be described in order.

3.1 Experiments 1a and 1b: Pattern documentation [maintenance]

Experiment 1a and its variation 1b compared the speed and correctness of maintenance on two pairs of programs: one of each pair using design patterns and documenting their use, the other with identical code but the documentation of pattern uses removed. Even the latter programs were so thoroughly documented that quantitatively the removal of the design pattern documentation should not make any difference unless some specific quality of design pattern documentation is indeed important. The full documentation of the experiment is [7, 9, 8].

Question: Is claim **CM** true? Actually, our experiment avoids a severe problem for testing this claim: If one compared programs with patterns to different programs without, it would be unclear whether the results originate from communication improvements or from structural program differences. In contrast, our experiment uses the *same* program in both groups, only the documentation with respect to the patterns is different.

Subjects: For 1a, we taught an optional six-week intensive graduate lab course on Java; the best 58 of initially about 100 participants qualified as experimental subjects. In the course and assignments, we taught a small number of design patterns as an illustration of good Java design style. For 1b, we repeated the experiment using similar programs with 22 participants of an undergraduate course on C++ and design patterns at Washington University, St. Louis.

Procedure: Experiment 1a was carried out on paper, whereas subjects of experiment 1b directly implemented on a workstation. Each participant worked on two different programs and wrote solutions for certain maintenance tasks. The groups were arranged to form a 4-group counter-balanced design.

Programs and tasks: Program *Phonebook* implemented a simple address book. The Observer pattern was used and two observer classes with different functionality were already implemented. The main task was adding a third observer, i.e., understand the specific Observer pattern instance given and implement a new observer class. The functionality differences between the three observer classes were nontrivial.
Program *And/Or tree* implemented trees of String nodes (leafs), String concatenation nodes (And) and String alternation nodes (Or). A Composite plus Visitor pattern was realized in the program to be used for arbitrary tree traversal routines and a single visitor class existed for computing the tree depth. The task was adding a second visitor class for computing the total number of alternations (from cross-product of Or nodes) of the tree. The task could be solved either by a single new visitor class or by adding one method to each node type class.

Results: For one of the programs (involving an Observer pattern), the group with pattern documentation was much faster than the other in 1a; see also Table 1. Refer to [9] for a discussion of the more complex 1b results. For the other program (involving a Composite and a Visitor pattern), the group with pattern documentation either had far fewer errors (1a) or required less time (1b). These differences are statistically significant.
We find claim **CM** supported and conclude that the communication improvement may become visible as either a productivity or a quality improvement, depending on the situation (program, maintainers, schedule pressure, etc). See [8] for further discussion of these results.

3.2 Experiment 2: Patterns versus alternative designs [maintenance]

Experiment 2 investigated whether using patterns is beneficial *at all* (aside from documenting them) and whether the difference depends on the level of pattern knowledge. The detailed description of the experiment is [10].

Question: Is the use of patterns beneficial (during maintenance) compared to simpler solutions of the same problem? This question includes aspects of claims **QE** and **CM** and it leads to the methodological problem of what is a fair alternative to a pattern for the comparison. We separately assess the question both with the same subjects before and after a pattern course, hence also investigate some aspects of claim **LD**.

Subjects: We taught a pattern course (two half days) to 29 professional software engineers of sd&m in Munich. The experiment was conducted in two parts before and after the course. All subjects were C++ pro-

Variable	mean with PD PD^+	mean w/o PD PD^-	means difference (90% confid.) I	significance p
total points	20.8	21.1	$-6.0\% \ldots +3.3\%$	0.35
relevant points	16.1	16.3	$-8.0\% \ldots +4.0\%$	0.35
number of correct solutions	17 of 36	15 of 38		
time (minutes)	51.5	57.9	$-22\% \ldots +0.3\%$	**0.055**

Table 1: Results for the Observer task of Experiment 1a. Columns are (from left to right): name of variable, arithmetic average PD^+ of sample of subjects provided with design pattern information (PD), ditto without, 90% confidence interval I for the difference $PD^+ - PD^-$ (measured in percent of PD^-), significance p of the difference (one-sided). p describes the probability that the observed differences occurred merely by chance alone. The "points" judge the correctness of a solution. "Relevant points" are the points for the pattern-relevant subtasks only. Many distributions were distinctly non-normal, therefore I and p were computed using the percentile method after 10000 trials of Bootstrap resampling [4].

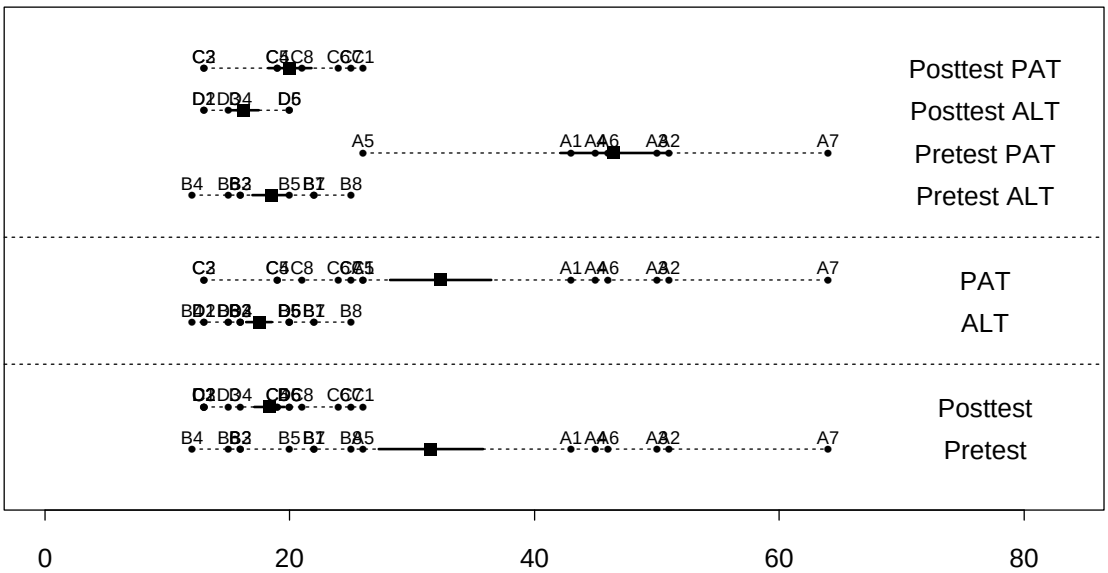

Figure 1: Time in minutes required for program *Stock Ticker* work task 1 of experiment 2, separately for the design pattern (PAT) version and the simpler alternative (ALT) version. Each dot marks one subject, the square is the arithmetic mean, the non-dotted line next to the square indicates plus/minus one standard error of the mean. If two such lines overlap, the difference of the respective means is statistically not significant. 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 ALT combined. Likewise, the bottom area (2 lines) shows PAT and ALT groups combined.

grammers with an average of 4.1 years of professional experience. Most had little previous experience with patterns, half of them none at all.

Procedure: Overall there were four different program pairs, each pair with two or three maintenance tasks to be solved. Each subject worked on two programs before the pattern course (pretest) and on the other two after the course (posttest). Each program pair consisted of two versions (each of which was used by half of the subjects): One version used design patterns. The other used a different alternative solution. The alternative was a reasonable design because some part of the functionality or flexibility of the design pattern solution was not required for the program at hand. The answers were written on paper.

Programs: Program *Stock ticker* contained an Observer pattern. Its purpose was multiple alternative dynamic views on a stream of stock trade events. The alternative version was similar but had no registration mechanism and the notification was hardcoded.

Program *Communication channels* contained a Decorator pattern. Its purpose was the transparent addition of encryption, compression etc. In the alternative version, all of this functionality was realized in a single class and could be selected by switches.

Program *Graphics library* contained a Composite and an Abstract Factory pattern. Their purpose was transparent grouping of graphical objects and transparent handling of multiple device types. The alternative solution had only modest structural differences.

Program *Boolean* contained a Composite and Visitor pattern. Their purpose was representing and prettyprinting Boolean Formulas consisting of And, Or, Xor and Not terms and variables. The alternative ver-

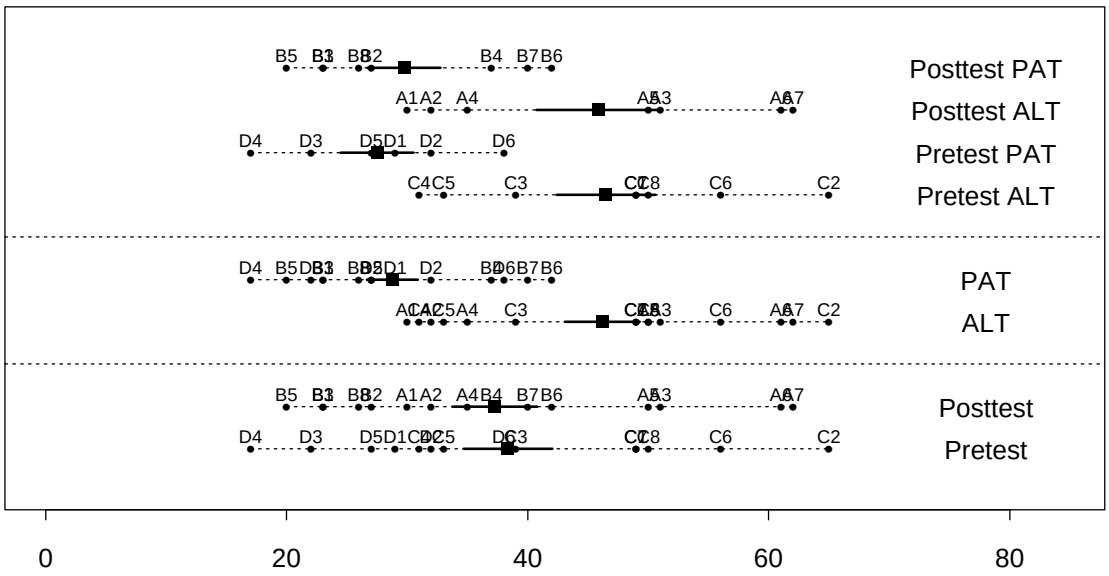

Figure 2: Time required for program *Communication Channels* work task 1 of experiment 2.

sion replaced the Visitor by methods distributed over all term classes.

There were multiple maintenance tasks for each program that are described in detail in [10].

Results: In the specific setting of the experiment, all three possible outcomes occurred: For the program 'Stock Ticker', involving an Observer, the pattern program version took longer to maintain, in particular when the subjects had low levels of pattern knowledge (pretest); see Figure 1. For the programs 'Graphics Library', involving a Composite and an Abstract Factory, and 'Boolean Formulas', involving a Composite and a Visitor, the pattern version and the alternative solution were just as good. For the program 'Communication Channels', involving a Decorator, the pattern solution was clearly superior, *even* in the pretest; see Figure 2. The differences mentioned are statistically significant.

This may seem like an unsatisfying result. However, its interesting feature is the fact that most of the results could be correctly predicted, e.g. by analyzing the locality of interactions within the program or the locality of anticipated changes. Only for the Visitor we had expected a negative effect which did not occur, but still we could explain it after the fact. This means that (1) generally, software engineering common sense, when carefully applied, is a suitable predictor of the relative usefulness of design patterns versus alternative solutions; (2) design patterns are useful, but are no cure-all; and (3) there are cases where our current understanding of designs and their comprehension misleads us into expecting difficulties at the wrong places. This point (3), together with the flexibility of the Gang-of-four design patterns, suggests to always use design patterns, except when specific reasons can be identified why a particular alternative solution will be better.

We are currently working on a statistical model of our results that quantifies the relative contributions of various variables (e.g. personal experience) to the results.

3.3 Experiment 3: Communication between developers and maintainers [design phase]

We are currently planning a third experiment which will investigate the usefulness of patterns to improve communication during the design phase (Claim **CD**). Anecdotal evidence suggests that patterns are particularly beneficial in this situation.

We consider the design of a program extension, where one team member knows the previous design and the others do not, hence involving an additional aspect of claim **CM** as well. The team has to develop a suitable design together.

The experimental setup might be: Each team consists of 2-3 subjects. To one person of each team we hand out a given design D, including a description of the design rationale, so that s/he can familiarize himself with the design so as to approximate the knowledge of the original developer. Then a meeting of the team is scheduled where an extension task T is to be solved. In a first step the developer explains the design to the other team members. Then the whole team discusses (and decides between) alternative design solutions to the given task. The explanation of the developer and the discussion will be videotaped, transcribed, and analyzed for typical communication events, communication breakdowns, etc. We expect that the contributions of claims **CD** and **CM** can be distinguished in our observations making the experiment even more useful.

The independent variables in this experiment are:

- patterns in persons: the members of the team may have general design pattern knowledge ($K = true$) or not ($K = false$) Alternatively, one could have

group	first step $K = false$	second step	third step $K = true$
Group 1	solve work package P	take pattern course	solve work package N
Group 2	solve work package N	take pattern course	solve work package P

Table 2: Suggested experiment design for experiment 3 ("communication between designers")

different levels of pattern knowledge within the team.

- patterns in program: the original design D may involve design patterns or not.
- patterns in task: If D contains patterns, the extension task T may directly involve these patterns or not.

One may be interested in various combinations of these variables. We choose to vary the first variable by a pretest/posttest design, teaching a course on design patterns in between. We vary the second and third variable together by using two different combinations of program+task: For work package P, the program and the extension task involve patterns, whereas for work package N, both do not involve patterns.

This results in the experiment design shown in Table 2. New teams will be composed for each task to reduce familiarization to other team members in the experiment.

Dependent variables are for instance

- the frequency and cause of communication breakdowns or misunderstandings,
- the means of recovering from these,
- the frequency and context of use of different communication styles, media, technical terms, target program concepts etc., and
- the local purpose of these uses: either establishing a communication episode or continuing (or reestablishing) a previous one.

In contrast to experiments 1 and 2, many dependent variables are of a qualitative nature or require context consisting of qualitative variables.

The following issues still have to be resolved:

1. Learning effects may arise from getting familiar to the discussions. These learning effects may dominate the intended experiment effect. How can we avoid this?
2. How can we objectively compare the design results?
3. How can we find a suitable work package pair? P and N must be similar with respect to the design communication, except for the occurrence of design patterns in P.
4. How can we relate between N and P those aspects of the communication behavior that are *qualitatively* different?

4 Conclusion

In order to avoid the mistake made with object-orientation, the software engineering research community should thoroughly test the claims made for software design patterns. Practically speaking, however, this is rather difficult.

The art of empirical software engineering is to produce useful scientific insight despite the tight practical constraints in which it has to work. In particular for controlled experiments the qualitative properties (task characteristics) of a software engineering situation must be preserved although its quantitative aspects (task size) have to be scaled down extremely far.

This article describes how the first reproducible scientific results on the properties of software design pattern usage have thus been obtained. This research has a good price/performance ratio, because useful practical advice can be derived from its results: Experiments 1a and 1b showed that carefully documenting pattern usage is highly recommendable because it pays off well during maintenance. Experiment 2 proved, first, that design patterns can be beneficial even when an alternative solution appears to be simpler but, second, that unsuitable application can also be harmful in other situations. The resulting practical advice calls to apply common sense when using design patterns instead of using them in a cookbook fashion (but, if in doubt, to prefer them because of their high flexibility).

Many more experiments (and studies of other types) will be necessary for completing our understanding of the design pattern idea.

Further information

Detailed information about all our experiments is available from http://wwwipd.ira.uka.de/~exp. This site in particular contains the experimental materials and raw result data in source form.

Acknowledgements

Michael Philippsen and Walter Tichy worked with us in experiments 1 and 2, respectively. Ernst Denert initiated and Peter Brössler organized experiment 2 at sd&m. Most of all we thank all of our experimental subjects, without whom...

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] 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.

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

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

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

[6] Adam A. Porter, Harvey Siy, Carol A. Toman, and Lawrence G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. In *Proc. Third ACM Sigsoft Symposium on the Foundations of Software Engineering*, 1995.

[7] 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.

[8] 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/.

[9] 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.

[10] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. on Software Engineering*, 1998. To be submitted. http://wwwipd.ira.uka.de/~prechelt/Biblio/.

[11] 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").

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