Appeared in "Proc. ICSE Workshop on Process Modeling and Empirical Studies of Software Evolution" pp. 72-76, Boston, MA, May 18, 1997.

Documenting Design Patterns in Code Eases Program Maintenance

Lutz Prechelt, Barbara Unger, Michael Philippsen
(prechelt,unger,phlipp@ira.uka.de)
Fakultät für Informatik, Universität Karlsruhe
D-76128 Karlsruhe, Germany

Abstract

Software design patterns are a promising idea with many advocates. While subjective reports of their usefulness are available, scientific proof is still missing.

We consider the case of programmers using design pattern documentation (in the form of comments in the source program) during maintenance. Is such pattern documentation (PD) helpful for understanding a program more quickly and designing better solutions for given maintenance tasks?

We describe a controlled experiment for investigating this question, present its results, and conclude that design pattern documentation can speed up program changes as well as improve their quality.

1 Background

Design patterns [3] are a promising approach to simplifying the design and construction as well as the understanding and evolution of software systems. Design patterns are prepackaged design solutions on the level of a few classes ("micro architectures"). They provide a solution approach for a class of problems. Besides the solution(s), design patterns also discuss the context where they are appropriate and the advantages and disadvantages of the solution variants.

Note that the new idea is not in the individual patterns itself (which are *not* meant to be new, but known and time-proven solutions), but in their *explicit* documentation and use as design building blocks.

The idea of design patterns is appealing and practitioners report subjectively that design patterns (1) simplify communication between designers by providing a concise common vocabulary, (2) can be used to record and reuse best practices, and (3) "capture the essential parts of a design in compact form" [1].

However, no rigorous test has yet been presented that design patterns are useful. This paper is a report on the first controlled experiment that investigates the benefits obtained from explicit use (here: documentation) of design patterns. We shortly describe our experiment and its main results. You will find an extensive description and evaluation of the experiment in [4].

2 Our Experiment

2.1 Hypothesis and Approach

The purpose of the experiment was to test whether well-documented programs could benefit (during program evolution) from additional documentation of the design patterns used.

We presented each subject with two programs and asked them to outline how certain extensions to these programs could be made. One program had its design patterns documented, the other had not. We asked the subjects to provide not the first solution they found, but the one that would result in the cleanest software structure, and to take as much time as they wished.

2.2 Subjects and Environment

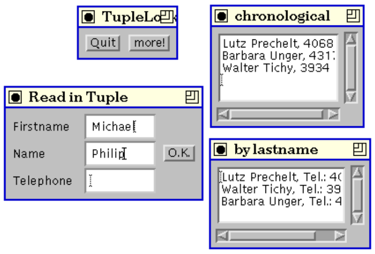
The 74 subjects were male Informatics students working towards the *Diplom* (M.Sc. in CS), 64 of them already had a *Vordiplom* (B.Sc. in CS). They had taken a 6-week lecture and lab course on Java, AWT (the Java GUI library), and design patterns just before the experiment (5 lectures of 90 minutes and about 50 hours of lab exercise). On average, their previous programming experience was 7.5 years using 4.6 different languages with a largest program of 3510 LOC. Before the course, 69% had previous nontrivial experience with object-oriented programming, 58% with programming GUIs.

The experiment was performed on January 25, 1997, in a single session of 2 to 4 hours. The programs were available to the subjects as pretty-printed listings. The subjects had to write their solutions on paper.

2.3 Programs Used

Both programs were written in Java using design patterns and were thoroughly commented. Program 1 (*Element*) was a library for handling And/Or-trees of strings and a toy application of it. It had 7 classes, spanned 7 printed pages (362 source lines, 133 of which were comments), and used the Composite and the Visitor design pattern [3].

Program 2 (*Tuple*) was 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 on the right. It had 11 classes, spanned 10 printed pages (565 source lines, 197 of which were comments), and used the Observer and the Template Method design pattern [3].



2.4 Independent Variable and Experiment Controls

The independent variable in this experiment was the presence or absence of design pattern documentation (PD) in the comments of the source programs. One of the programs given to each subject had its design patterns documented in addition to the normal comments (31 lines of PD added to *Element*, 20 lines added to *Tuple*), the other had no such additional documentation.

We balanced across the subjects the order of the two programs, the order of having and not having PD, and the combination of both, i.e., we used a counterbalanced experiment design [2].[1] Furthermore, we also balanced the four resulting groups for expected subject ability, measured by the number of points each subject received in the lab course, using stratified random sampling. The experiment was conducted semi-blindly, i.e., the subjects did not know in advance whether a program would contain PD or not, but there was no placebo.

2.5 Tasks

For *Tuple* each subject had to perform the following 5 subtasks: (1,2) Finding two spots for small program changes (output format change, window size change), (3) creating an additional Observer class similar to an already existing one using a Template Method[2], (4) creating and installing such an Observer similar to the two existing Observers, (5) creating an additional observer class similar to an already existing one not using a Template Method.

For *Element* each subject had to perform the following 4 subtasks: (1) Finding the right spot for a particular output format change, (2) giving an expression to compute the number of variants represented by a tree, (3) creating an additional Visitor class for computing the number of variants more cheaply, similar to an already existing class computing depth information, (4) creating such a Visitor and printing its result.

Subtasks 1 and 2 of both *Tuple* and *Element* are independent of PD; for the other subtasks, PD is expected to help. For the class creation subtasks, only the interface of the class needed to be written; the actual implementation was not required.

2.6 Measurements

For each task of each subject we measured the time taken by handing out and collecting the experiment materials incrementally (per task). It is unclear how the time spent for general program understanding could be distributed among the subtasks, so no subtask time information is available. For each subtask, we graded the answers according to the degree of requirements fulfillment they provided. The grades were expressed in points, much similar to a written exam (maxima: 2+3+8+4+6=23 points for the subtasks of *Tuple* and 2+2+8+3=15 points for those of *Element*). Furthermore, we recorded an error class code for each error made.

3 Results and Conclusions

The design of this experiment was *extremely* conservative; many design decisions biased the experiment towards *not* showing any effects from adding PD:

1. The subjects knew they would participate in an experiment "about design patterns", so they were very motivated to find patterns in the programs.
2. Furthermore, the subjects knew only few design patterns well; so even without PD they roughly knew what they could expect.
3. The programs were rather small, so even without PD the subjects could achieve good program understanding within a reasonable time.

[1] That is, group 1 (19 subjects) performed first Element with PD, then Tuple without PD; group 2 (18 subjects) performed Tuple with PD, then Element without; group 3 (18 subjects): Element without, then Tuple with; group 4 (19 subjects): Tuple without, then Element with.

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

Variable	mean with PD P^+	mean w/o PD P^-	means difference (90% confid.) I	significance p
Tuple:				
time (minutes)	51.5	57.9	$-22\% \ldots +0.3\%$	**0.055**
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
points per hour	27.6	24.7	$-6.1\% \ldots +29\%$	0.14
Element:				
time (minutes)	58.0	52.2	$-3.0\% \ldots +24\%$	**0.094**
points	11.1	10.4	$-8.2\% \ldots +22\%$	0.23
relevant points	8.5	7.8	$-7.7\% \ldots +23\%$	0.20
points per hour	12.8	14.7	$-34\% \ldots +7.2\%$	0.14
time, correct solutions	46.9	45.4	$-22\% \ldots +27\%$	0.41

Table 1: (left to right:) Name of variable, arithmetic average P^+ of sample of subjects provided with design pattern information, ditto without, 90% confidence interval I for difference $P^+ - P^-$ (measured in percent of P^-), significance p of the difference. "relevant points" are points excluding subtasks 1 and 2. As neither the time nor the point distributions were normal, I and p were computed using resampling statistics with 10000 trials [5].

4. Due to the small program size, the pattern density in the programs was quite large. Therefore one could find the patterns quickly even if they were not documented.
5. The programs were thoroughly commented, not only on the statement level, but also on the method, class, and program levels. Thus, the subjects had sufficient documentation available for program understanding even without PD.
6. All pattern-relevant tasks required adding functionality similar to existing functionality. So even without PD, the new parts could be derived from analogous classes or statements.

Given these circumstances, we expect performance advantages from having PD to be much more pronounced in real situations than in our experiment. Therefore, any significant result found in the experiment is a strong sign that PD in program documentation is really useful.

We found the following significant results (summarized in Table 1):

Program *Tuple*: Subjects with and without PD solved the tasks similarly well, but the former were significantly faster ($p = 0.055$). With respect to pattern-relevant error classes, subjects with PD forgot to register their new Observer more often in subtask 4 ($\chi^2 = 1.18$, $p = 0.14$), while subjects without PD more often made inheritance errors in subtask 5 ($\chi^2 = 1.16$, $p = 0.14$). Other differences in performance were not significant.
Interpretation: In the given context, the tasks were simple enough that they were solved rather well even without the PD. However, PD allowed subjects to find their way through the 11 classes more quickly.

Program *Element*: Subjects with PD took significantly *longer* to solve the tasks ($p = 0.094$), yet still overall they had no significantly better solutions, i.e., more points ($p = 0.23$) or "relevant points" ($p = 0.20$). However, the size of the subgroups with completely correct solutions was 15 with PD and only 7 without; this difference is significant ($\chi^2 = 3.55$, $p = 0.03$). The task time difference is completely insignificant ($p = 0.41$) for the best 7 subjects (according to lab course performance) of each group that had completely correct solutions.
Interpretation: For this smaller program, PD increased the time required for finding a correct solution; probably because the problem could more easily be solved by imitating a class already present in the

program than by understanding the design patterns used and constructing a solution: 42% of Element subjects without PD said they did not search for patterns at all and 42%/47% did not recognize the Visitor/Composite (Tuple: 32% non-searching and 32%/24% missed Observer/Template Method). However, the imitation approach had one pitfall and promptly significantly more subjects with PD found a completely correct solution than without.

As we see, PD helped to improve either task completion time or solution quality, depending on the kind of task and program. In a quantitative sense, these results are not dramatic. But given the conservative design of our experiment, they indicate that documenting design patterns in software can improve program maintenance.

Interestingly, according to our postmortem questionnaire, the presence or absence of PD influenced neither the perceived task difficulty nor the subjects' perceived ability for concentration on the task — although on average the subjects reported to concentrate less well on their second task than on their first.

It is unclear how these results will scale to more experienced software engineers and to programs of industrial size and complexity, but we believe that in situations where design patterns are relevant during maintenance, one would see benefits from PD that are at least as large as in the experiment. Therefore, we recommend that when design patterns are used they should be explicitly documented in the program code.

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

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

[4] Lutz Prechelt. An experiment on the usefulness of design patterns: Detailed description and evaluation. Technical Report 9/1997, Fakultät für Informatik, Universität Karlsruhe, Germany, April/May 1997. ftp.ira.uka.de/pub/techreports.

[5] Julien L. Simon. *Resampling: The new statistics.* Duxbury Press, Belmont, CA, 1992. http://www.statistics.com.

Biography

Lutz Prechelt is a research associate with research interests in empirical software engineering (in particular design patterns and personal process improvement), benchmarking and quantitative methods, and compiler construction for parallel processing.

Barbara Unger is a Ph.D. student whose interests are in empirical software engineering.

Michael Philippsen is a research associate with interests in compiler construction, in particular optimizations for object-oriented parallel and distributed programming, and in software engineering, in particular design patterns.