Accepted for 'IEEE Trans. on Software Engineering', July 28, 2001

Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance

Lutz Prechelt, Barbara Unger, Michael Philippsen, Walter Tichy
Fakultät für Informatik, Universität Karlsruhe
D-76128 Karlsruhe, Germany
Phone: +49/721/608-3934, Fax: +49/721/608-7343
Email: prechelt,unger,phlipp,tichy@ira.uka.de
WWW: http://wwwipd.ira.uka.de/EIR/

Abstract

Using design patterns is claimed to improve programmer productivity and software quality. Such improvements may manifest both at construction time (in faster and better program design) and at maintenance time (in faster and more accurate program comprehension). This paper focuses on the maintenance context and reports on experimental tests of the following question: Does it help the maintainer if the design patterns in the program code are documented *explicitly* (using source code comments), compared to a well-commented program without explicit reference to design patterns?

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 use of design patterns was documented explicitly or not. The experiments thus tested whether pattern comment lines (PCL) help during maintenance if patterns are relevant and sufficient program comments are already present. It turns out that this question is a challenge for the experimental methodology: a setup leading to relevant results is quite difficult to find. We discuss these issues in detail and suggest a general approach to such situations.

The experiment was performed with Java by 74 German graduate students and then repeated with C++ by 22 American undergraduate students. A conservative analysis of the results supports the hypothesis that pattern-relevant maintenance tasks were completed faster or with fewer errors if redundant design pattern information was provided. Redundant means that the information carried in pattern comments is also available in different form in other comments.

The contribution of this article is twofold: It provides the first controlled experiment results on design pattern usage and it presents a solution approach to an important class of experiment design problems for experiments regarding documentation.

Keywords: controlled experiment, design pattern, comments, documentation, maintenance

1 Introduction

A software design pattern describes a proven solution to a software design problem with the goal of making the solution reusable. Design patterns 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 [9] (nicknamed the "Gang of Four Book"). Shortly thereafter, additional patterns were reported by Buschmann et al. [4]. The book by Shaw and Garlan [19] also provides a wealth of patterns for software architecture. Annual workshops are being held [6, 25, 14] to promote pattern mining and a consistent style of reporting patterns. Pattern papers are published 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, identifying 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 increase their design skills significantly 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.

As yet, there exists no coherent theory that would explain these advantages specifically in the context of design patterns.

1.1 Our experiments

The experiments reported here represent the first attempts at formally testing some aspects of the above claim 4. Our 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* (using source code comments), compared to a well-commented program without explicit 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; the subjects then 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 commented in detail but the subjects in the control group receive no explicit information about design patterns in the program; they may derive pattern information from other program comments or not at all. The experiment group, by contrast, receives the same program with the same comments plus a few additional lines of comment (called *pattern comment lines* or PCL) that describe pattern usage where applicable. Note that the experiment group thus has slightly *more* comments in their program. This might threaten the validity of the experiments. We will discuss and resolve this important methodological issue in Section 3. 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 different programs about 6 to 10 printed pages in length.

1.2 Experiment rationale

Note that the experiments do not test the presence or absence of patterns, but merely the documentation of patterns. This question is much cleaner, because it is unclear what alternative program design should be used for comparison when testing the presence or absence of patterns.

Why do we think that adding PCL is a useful proposition? Most theories of program comprehension state that the comprehension process is driven by hypotheses [3, 13] formed and validated during the understanding process. (A nice overview of this view of program comprehension can be found in [26]. Further recent research papers can be found in [28, 29, 30, 31].) Once a program is understood, its meaning is represented as a hierarchy of hypotheses. Without prior knowledge, systematic program understanding must work bottom-up [15], inducing the meaning and purpose of program parts step-by-step, starting from individual declarations and statements. However, a completely bottom-up approach is unrealistic for larger programs. Therefore, if a programmer finds hints (so-called beacons) to familiar kinds of structures within the code, he or she will switch to a top-down approach of understanding that allows for generating and validating hypotheses much quicker. Beacons can be structural (such as code idioms), but more often they are simply names of program entities [10].

If this view is correct, PCL are probably a very powerful aid to program understanding, because each description of a design pattern usage directly leads to a large-grain hypothesis that spans multiple classes and that is relatively easy to verify top-down. In this case it would be a highly efficient engineering practice to always document the use of design patterns by PCL because writing PCL requires only little effort. Slightly different views of program comprehension discussed in the next section provide further support for the idea that adding PCL may be highly worthwhile.

1.3 Other related work

The program comprehension theory presented by Brooks [2] suggests that PCL may be a highly effective sort of comment, because it specifically aims at bridging different "knowledge domains".

A different view is provided by the notion of *delocalized plans* [21] introduced by Soloway et al. Their experiments showed that delocalized plans account for much of the effort during program understanding. Wilde et al. [27] argue that object-oriented programs are full of delocalized plans due to inheritance and delegation. Design pattern instances are delocalized plans as well, as a single design idea is spread over multiple classes and methods. From this perspective, PCL may be helpful because it links the parts of important delocalized plans.

The empirical work of Shneiderman and Mayer [20] confirms several basic assumptions about comments: mnemonic names and comments both are useful for program understanding; higher-level comments are more useful than comments on a level of abstraction close to individual program statements. We do not know of any experiments comparing different forms of program comments in the context of object-oriented programs.

All of the work mentioned so far is not concerned with design patterns per se and are have in fact been published long before the notion of software design patterns became popular. Published evidence about the effectiveness of design patterns is still scarce. Case study reports and anecdotal evidence of positive effects can be found for instance in [1, 9]. Part of the program maintenance literature is loosely relevant to pattern effectiveness, but we have found no empirical work that specifically addresses 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.

We are not aware of any reports on the usage density of PCL in software containing design pattern uses. However, it appears that in thoroughly designed APIs of libraries and frameworks at least a surrogate to PCL becomes increasingly common: naming conventions that indicate the usage of some of the more frequent design patterns; for instance class name components such as listener, event, composite, container, observer in Sun's Java Development Kit (JDK 1.3) etc.

1.4 Structure of the article

The next section describes the design and implementation of the experiments including a statement of the hypotheses, a description of the subjects' background, a description of the program (including program comment examples) and tasks, and a discussion of possible threats to the internal and external validity of the experiments. Section 3 discusses the methodological problems caused by adding comments and how to handle them in similar situations. Section 4 presents the results, Section 5 summarizes and raises questions for future research.

We provide only a short description of the programs and tasks. Detailed information, including all original experiment materials such as task descriptions and source program listings, is available in [8, 17, 18].

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. Also the strengths and weaknesses of their data are not in the same spots, so that the experiments complement one another quite nicely. We therefore describe the experiments separately and refer to them as UKA and WUSTL, respectively. Where appropriate, we provide 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, which we straightforwardly derived from claims found in the design pattern literature [1, 9]:

Hypothesis H1: By adding PCL, pattern-relevant maintenance tasks are completed faster.

Hypothesis H2: By adding PCL, fewer errors are committed in pattern-relevant maintenance tasks.

Speed of task completion is measured in minutes. The number of errors are quantified by assigning points (see Section 2.6) and by counting correct solutions.

2.2 Subjects and environments

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. The course included practical implementations of programs involving the following design patterns: Composite, Visitor, Observer, Template Method, Strategy ⟨Wrapper, Adapter, Template Method, Bridge, Singleton, Facade, Strategy, Factory Method, Visitor, Interpreter, Builder⟩. On average, the subjects' 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 some previous experience with object-oriented programming, 58% ⟨50%⟩ with programming GUIs.

The subjects were knowledgeable about design patterns, as indicated by a pattern knowledge test conducted at the start of each experiment [17, 18]. For those patterns that were relevant in the experiments, 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 PCL were added in the version with PCL. (Section 3 discusses why adding PCL is an acceptable and meaningful manipulation.) *And/Or-tree* uses the Composite and the Visitor design pattern [9].

Program *Phonebook* is a GUI program for reading tuples (name, first name, phone number) entered by the user and showing them in different views

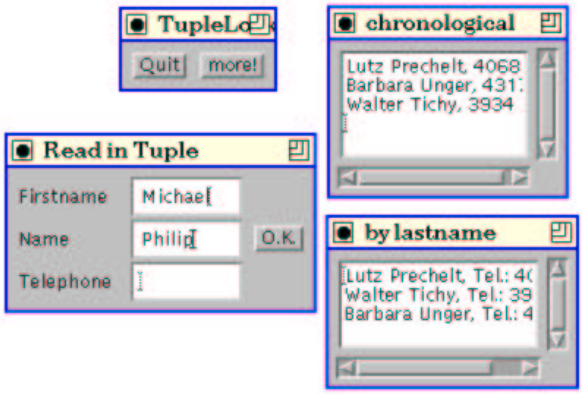


Figure 1: *Screenshot of* UKA *Phonebook program*

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 PCL were added in the version with PCL. *Phonebook* uses the Observer and the Template Method design pattern [9]. See Figure 2 for examples of PCL sections and other program comments.

[8, 17, 18] contain the full commented source code of the programs.

2.4 Experiment controls and group sizes

The independent variable in both experiments was the presence or absence of design pattern comment lines (PCL) in the comments of the source programs.

We used a counterbalanced experiment design [5] with random subject assignment, see Table 1: The first variable is the order in which a subject receives the two programs. One of those programs was supplied with PCL, the other without. This design results in a second variable, i.e., the order of PCL and non-PCL: first with, then without PCL, and vice versa. The combination of the variables results in four groups. The subjects did not know in advance whether a program would contain PCL or not; they did not even know that the existence of PCL would be a treatment variable.

2.5 Tasks

For *And/Or-tree*, each subject received a task consisting of the following 4 subtasks: (1) Find the

```
365  abstract class TupleDispA extends TupleDisplay {
       ...
403    /** implements adding a new Tuple.
404        First select() is used to test, whether the Tuple should be added
405        at all, then mergeIn() moves it to the right place in the
406        presentation using compare() and format() converts it into a String
407        of the desired display format.
408        *** DESIGN PATTERN: ***
409        newTuple() together with its auxiliary method mergeIn() forms a
410        **Template Method**. The empty spots that are filled in subclasses
411        are the methods select(), format(), and compare().
412    */
413    synchronized void newTuple(Tuple t) {
         ...
419    }
       ...
473  }
     ...
477  /**
478      NTTupleDisp2 displays NTTuple, where
479      1. Tuples with an empty telephone number are left out and
480      2. Tuples are sorted by (last)name
481      Using Tuple objects of other Tuple types results in
482      ClassCastException.
483      *** DESIGN PATTERN: ***
484      NTTupleDisp2 completes the **Template Method** newTuple()
485      of TupleDispA
486  */
487  final class NTTupleDisp2 extends TupleDispA {
       ...
513  }
     ...
517  /**
518      Main program. Generates a main window with two buttons, one
519      Tupleset and two TupleDisplays. One of the buttons creates an
520      NTTuple and adds it to the Tupleset.
521      There is no static type safety between the actual Tuple type
522      stored in the Tupleset and the Tuple type that is expected by
523      the TupleDisplays.
524      *** DESIGN PATTERN: ***
525      The two TupleDisplays are registered as observers at the Tupleset.
526  */
527  public final class TupleMain extends Frame {
       ...
573  }
```

Figure 2: *PCL example: Three (out of four) comments containing PCL sections from the* UKA *Phonebook program, with original line numbers. Added PCL sections are lines 36-42 (in a 33-line global program comment not shown here), 409-411, 484-485, and 525. Note that since the PCL header line does not contain semantic information, it was not considered part of the PCL although its existence may well be quite useful during program understanding. The program contains a total of 197 lines (or 35% of all lines) of normal comments. Note that the original program was in German with one more pattern comment line due to different line breaks.*

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 pattern-relevant. Subtask (4) was not explicitly required in WUSTL.

For *Phonebook*, each UKA subject received a task consisting of 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, (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 pattern-relevant.

Of course, none of the task descriptions mentioned any of the design patterns at all.

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, subtask (2) was not required in WUSTL and subtask (4) was stated as a part of (3) rather than as a separate subtask.

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 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 un-

Table 1: *The four experiment groups and their sizes. 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 PCL, then perform Phonebook without PCL" and so on.)*

	first with PCL then w/o PCL	first w/o PCL then with PCL
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

clear how the time spent for general program understanding could be distributed among the subtasks, so no subtask time information was collected. For each subtask, the same rater graded all 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*; see [17, 18] for the rating scales.

2.7 Threats to internal validity

All relevant external variables (subjects' programming experience, aptitude, motivation, environmental conditions, etc.) were equalized between the groups by randomized group assignment. Should bad luck nevertheless have produced groups with unbalanced subject ability, the counter-balanced experiment design has put the more capable subjects once into the experiment group and once into the control group.

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. The tendency to give up was relatively strong, because the experiment was the very last event of the semester for these students and they told us that had to catch their busses and planes etc. Four students gave up on both tasks. It would have been nice had mortality been lower, but the results are still meaningful, because mortality occurred almost exactly as often in the groups with PCL as in those without PCL. When ignoring incomplete tasks entirely, the mortality does therefore not bias the results. See Table 1 for the resulting group sizes; there was no mortality in UKA.

The total working time of the subjects exhibits an almost perfectly symmetric distribution. Hence, there is no evidence that slower subjects hurried because of others having finished before them although all subjects worked in the same room at the same time.

Our experiment setup lacked an acceptance test. As a consequence, the working time and solution quality can only be judged together, since short working times may reflect unrepaired defects in a solution. This is not a threat to validity, but complicates the discussion of the results in Section 4.

Finally, although at least for UKA the number of subjects is comparatively large, the resulting data is still often too scarce for undebatable conclusions.

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 maintenance 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 either have less need for PCL or they may be able to exploit PCL more profitably than our student subjects.

Team work: Realistic programs are usually team work. Individual tasks during maintenance may also often be performed by more than one programmer. Such cooperation requires additional communication about the program. In this case, PCL may have further advantages, not visible in the experiments, because one of the major (purported) advantages of design patterns is a common design terminology [23].

Program size and complexity: Compared to typical industrial size programs, the experiment programs are rather small and simple, neatly designed, and well commented. This does not necessarily invalidate the results of the experiments, though. If a positive effect is found, increasing program complexity may magnify the effect, because having PCL provides program slicing information. For pattern-relevant tasks, PCL information points out which parts of a program are relevant and enables one to ignore the rest; such information may become more useful as program size increases, because more code can be ignored.

Program and task representativeness: It is unknown whether the programs and tasks used in our experiments are (or are not) representative of realistic maintenance situations. We have but one indication that our programs are at least not totally different from other programs constructed using design patterns: The ratio of the total number of classes in the program to the number of design pattern instances found in our programs ranges between 3.0 and 5.5. These values are comparable to those found for Java AWT (3.8) and NextStep (3.1) [11]. Our article does not claim anything about maintenance tasks that are not pattern-relevant. See the conclusion section for more discussion of pattern-relevant tasks in realistic programs.

Maintenance situations: Realistic maintenance situations will often be rather different from those found by our subjects. In particular, much larger and more complex programs and tasks may require making changes based on a much lower degree of overall program understanding than could be obtained for the small programs in the experiments. It is hard to say whether or when this will make PCL more useful or less useful than in the experiments. Furthermore, if programmers have to master a large design pattern repertoire, their understanding of individual patterns may be reduced and PCL may become less helpful.

Only repetition of similar experiments with professionals on real programs and real maintenance tasks can evaluate these threats. See the conclusion section for why we hope that our experiments estimate conservatively.

3 On the methodology of our experiment design

The main independent variable in these experiments is the presence or absence of pattern comment lines (PCL) in addition to the normal program comments. Some readers may argue that this makes the experiments uninteresting and irrelevant (although still valid), because design and hypotheses are impure: the observations cannot discriminate between effects from having a specific type of comments (PCL) and effects from simply having *more* comments.

3.1 The form/content conflict

Note that this situation will occur generically in many experiments that attempt evaluation of a specific form of an information source.

Each specific information source used during program development (such as PCL, UML diagrams, or any other) has both a certain *form* (syntax, appearance) and a certain *content* (semantics, meaning). In any experiment for evaluating some form F of an information source we ideally need an experiment condition for the control group that is identical to the experiment group condition in all respects (in particular the content) except for the form of the information source. However, an alternative form A will rarely lend itself to expressing the exact same content as F in a natural form. For example, one cannot express in a realistic manner *exactly each detail communicated by a UML diagram and nothing else* in a different notation. Therefore, we must either force the content into A (and thus end up with an unrealistic instance of A) or accept some difference in content in addition to the difference in form in our experiment conditions (and thus mix two effects).

We call this problem the *form/content conflict* in experiment design.

If it arises, it is usually impossible to eliminate the conflict, but it can be reduced so that the relevance of the experiment is no longer seriously threatened. The following three subsections argue why in our specific experiment design the conflict does not have a serious impact on the validity or utility of the

results. Subsection 3.5 then reformulates the train of thought of this argumentation as a general rule for the design of such experiments.

We now argue for our design, first, why purer hypotheses appear impossible to evaluate; second, why the experiments are useful as they are; and third, why not much of the observed effect is due to an increase in comment information content.

3.2 Why purer hypotheses appear impossible to evaluate

A hypothetical perfect experiment would only test whether the specific form (or type) of comments represented by PCL has an advantage over whatever one would have written instead before the invention of design pattern terminology. It would not change any other aspect of the program. In order to do this, the following aspects (among others) would have to be kept constant:

- the absolute information content of the comments (because this is what makes comments useful),
- the length of the comments in lines or words (because reading comments consumes time and concentration),
- the amount of repetition of information in the comments (because repetition may aid understanding or finding information or may make the reader unattentive),
- the understandability of the comments,
- the comments' potential for creating confusion (because comments can sometimes make a program *harder* to understand),
- the placement of the comments with respect to the program entities they refer to (because that determines how easy information can be found or related to the program).

Balancing *all* of these aspects in a program change is impossible. We have the following choices for the experimental design:

1. Compensate adding PCL by removing other comments in the PCL program version.
   But which other comments should we remove? We could easily balance the comment length, but there is no reproducible way to determine which comments are superfluous, useful, or crucial. We could not guarantee that we held the information content of the comments constant. Furthermore, we can not keep both comment length and the amount of repetition constant, as will be quantified in Section 3.4.
2. Compensate adding PCL by adding normal comments in the non-PCL program version.
   But what should these other comments say? Again, we could keep the comment length constant, but there is no reproducible way to give the exact same information content as in the PCL, other than the PCL itself; we cannot be sure not to confuse or annoy the user, etc.
3. Add PCL without any compensation. This will increase the comment length (small increase, see Section 2.3 and Figure 2), information content (small increase, see Section 3.4), and repetition of information (large increase, see Section 3.4).

We pick the third solution, because it is more realistic and well-defined and leads to more reproducible results.

3.3 Why the experiments are useful

A possible effect of simply having *more* comment is consciously included in our hypotheses. If we find a positive effect from adding PCL, the experiments suggest the engineering advice "Add PCL to your programs in addition to the commenting you usually do." This advice is useful even if it should be possible to obtain similarly positive results with other types of additional comments, because who could characterize, in program-independent terms, what these other comments should look like? For PCL, the characterization is straightforward: "Mention which classes, methods, and objects play which roles in a pattern"; see the example in Figure 2.

3.4 Why most of the effect is due to comment type, not information content

Are we merely (or mostly) measuring the effect of providing *more* information in the comments? As we show now, almost all of the information contained in the PCL is already present in the rest of the comments and is just repeated in a different form by PCL. Note that repetition as such can either improve program understanding (by useful redundancy) or slow it down (because less new information is found per line) [3]. See the conclusion section for our opinion on the present case.

We compiled a list of the *design information units* conveyed by the PCL and counted where and how often they were mentioned in the PCL and in the other comments.

Table 2: *List of the design information units provided by the PCL of the* UKA *And/Or-tree program (left hand side) and the* UKA *Phonebook program (right hand side). The histograms indicate the number of occurrences of each design information unit (with the important ones emphasized). Top histogram: occurrences in the PCL. Bottom histogram: occurrences in the rest of the comments.*

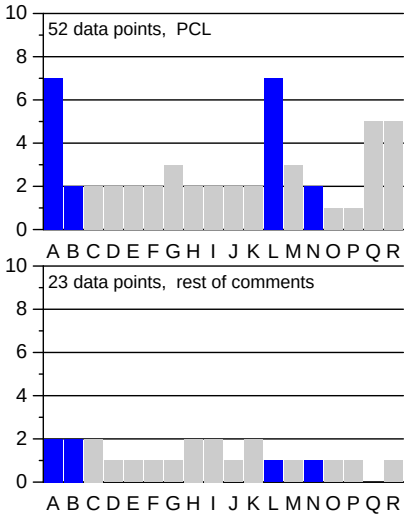
id	design information unit (UKA *And/Or-tree*)
A	There is an *element/container structure*
B	`Element` is the superclass of the *element/container structure*
C	`Element` is abstract
D	`AndElement` is a part of the *element/container structure*
E	`OrElement` is a part of the *element/container structure*
F	`StringElement` is a part of the *element/container structure*
G	There are multiple *container* classes
H	`AndElement` is a *container* class
I	`OrElement` is a *container* class
J	There is only one *element* class
K	`StringElement` is an *element* class*
L	There is an *iterator structure**
M	The *iterator structure* iterates over the *element/container structure**
N	`ElementAction` is the superclass of the *iterator structure**
O	`ElementAction` is abstract
P	`Depth` is a part of the *iterator structure*†
Q	`perform()` is a part of the *iterator structure*
R	`perform()` is a *dispatch method**

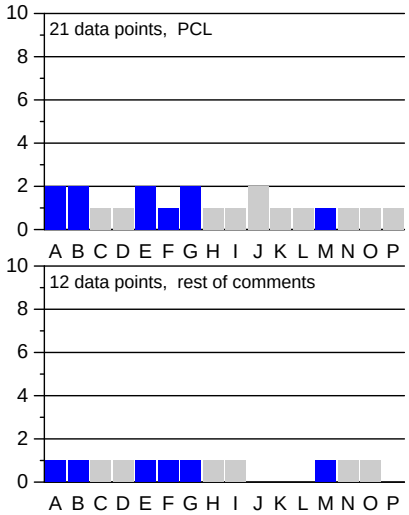
A, B, L, N are the most important information units for design understanding.
In design pattern terminology [9] *element/container structure* is *Composite pattern*, *container class* is *Composite class*, *element class* is *Leaf class*, *iterator structure* is *Visitor pattern*, and *dispatch method* is *Accept method*.
However, none of these design pattern terms is used in the non-PCL comments.

id	design information unit (UKA *Phonebook*)
A	There is a *model/view structure**
B	`Tupleset` is the *model* class*
C	`TupleDisplay` is the superclass of the *views*†
D	`TupleDisplay` is abstract
E	There is a *skeleton algorithm*
F	The *skeleton algorithm* varies selection, ordering, and formatting of tuples
G	`TupleDispA` contains a method with a *skeleton algorithm**
H	`TupleDispA` is a *view* class
I	`TupleDispA` is abstract
J	`newTuple()` is the *skeleton algorithm*
K	`mergeIn()` is an auxiliary method of `newTuple()`
L	`select()`, `compare()`, and `format()` are the *primitive operations* missing in the *skeleton algoritm*
M	`NTTupleDisp2` completes the *skeleton algorithm*†
N	There are two *view* class instances
O	There is one *model* class instance
P	The *view* class instances are registered with the *model* class instance

A, B, E, F, G, M are the most important information units for design understanding.
In the design pattern terminology of [9] *model/view structure* is *Observer pattern*, *model class* is *Subject class*, *view class* is *Observer class*, *skeleton algorithm* is *Template method*.

However, none of these design pattern terms is used in the non-PCL comments.





*This information is rather vague or ambiguous in the non-PCL comments.
†This information is distributed over several non-PCL comments.

Table 3: *Results for the And/Or-tree task. Columns are (from left to right): line number; name of variable; arithmetic average D^+ of subjects provided with design pattern information (PCL); ditto without PCL (D^-); 90% confidence interval I for difference $D^+ - D^-$ (measured in percent of D^-); significance p of the difference (one-sided). I and p were computed using Bootstrap resampling (a simulation-based statistical estimation technique [7]) with 10000 trials because many distributions were distinctly non-normal. "Relevant points" are the points for the pattern-relevant subtasks only.*

	Variable	mean with PCL D^+	mean w/o PCL D^-	means difference (90% confid.) I	significance p
	UKA, program *And/Or-tree*:				
1	relevant points	8.5	7.8	$-7.7\% \ldots + 23\%$	0.20
2	#corr. solutions	15 of 38	7 of 36		0.077
3	time (minutes)	58.0	52.2	$-3.0\% \ldots + 24\%$	0.094
4	— corr. only	52.3	45.4	$-11\% \ldots + 41\%$	0.17
	WUSTL, program *And/Or-tree*:				
5	relevant points	6.7	6.5	$-12\% \ldots + 19\%$	0.28
6	#corr. solutions	4 of 8	3 of 8		1
7	time (minutes)	52.1	67.5	$-43\% \ldots - 0.5\%$	0.046

For UKA *And/Or-tree*, this information is shown in the left half of Table 2. As we see in the histograms, almost all of the information present in the PCL is already present in the rest of the program comments, in particular those units (labeled A and L) that point to the existence of design patterns in the program.

Similar results, although somewhat weaker, hold for the UKA *Phonebook* program; see the right hand side of Table 2.

So the PCL information is almost entirely redundant and plausibly it is the specific *type* of the documentation that creates the effects we see: The same information might become clearer when expressed in PCL form. Hence, we expect our results to mainly show effects from specifically adding PCL and not effects from adding *any* documentation.

3.5 General methodological rule

We can now formulate the above three-step process for handling form/content conflicts as a general methodological recipe.

1. Analyze all conceivable designs and choose the one that results in the least differences in content without requiring unrealistic conditions for either group. This ensures there is no better design for this research question.
2. Demonstrate that the experiment result expected by your hypotheses would be interesting. Ideally, it leads to a practical software engineering rule. In this case, formulate this rule explicitly. This ensures the experiment is useful despite the content difference.
3. Argue why in your specific experiment setup at most a small part of the observed effect, if any, will be due to different content, rather than different form. This ensures the form/content conflict is unimportant for the chosen experiment design.

4 Results and discussion

This section discusses the results of both experiments as summarized in Tables 3 and 4. 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.

4.1 Results for *And/Or-tree*

The results are summarized in Table 3 and Figure 3. The first line of Table 3 indicates that the group with PCL averaged slightly more points in the pattern-relevant subtasks ("relevant points"). However, the p-value in the rightmost column indicates that the difference has a 20 percent chance of being accidental.

The time required with PCL appears to be larger than the time without PCL (line 3). However, this observation is misleading, because the number of correct solutions is much lower for the non-PCL group (Fisher exact $p = 0.077$) as shown in line 2. In real software maintenance, incorrect solutions would be detected and corrected, taking additional time not observed in the experiment. In contrast, the experiment's non-computerized working environment made it difficult for a subject to check

whether a solution was correct. Obviously, the time spent on incorrect work can not be sensibly compared to the time spent on correct work. Therefore, instead of comparing the time required by all subjects, we should compare only the time required by subjects with correct solutions. As a result, the confidence interval for the work time difference ranges from far positive to far negative. Hence, we should consider the work times to be essentially the same; see also Figure 3. The group with PCL is still a little slower, because it is a much larger fraction of the whole and hence presumably contained less capable subjects on average.

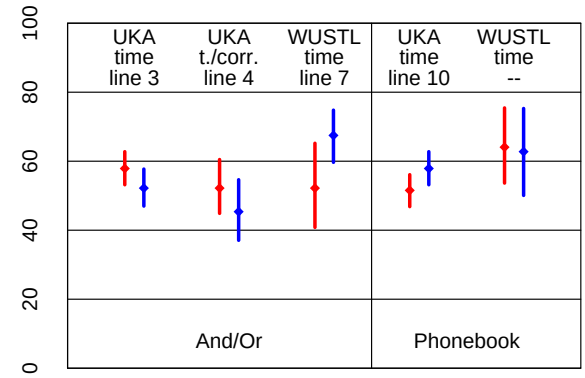


Figure 3: *Graphical display of the time entries (in minutes) from the indicated lines of Tables 3 and 4: The left plot of each pair represents the group with PCL, the right one the group without PCL. The dot marks the mean of the task completion time, the strip indicates a 90% confidence interval for the mean.*

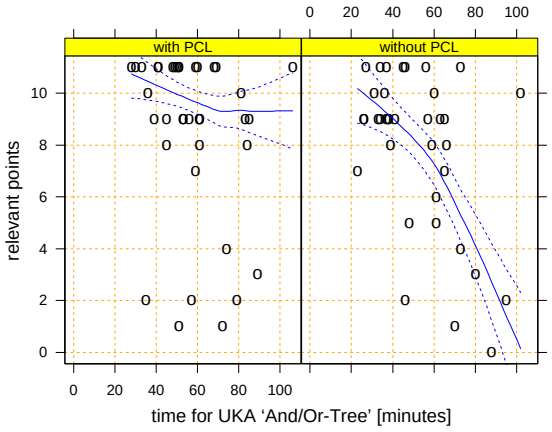


Figure 4: *Work time versus solution quality for the pattern-relevant part of* UKA *And/Or-tree task. The trend line is a Loess local linear regression [24] with spanwidth 1 and its 90% confidence interval.*

In terms of correctness alone, the much larger fraction of correct solutions in the PCL group suggests that PCL allowed for good solutions by less capable programmers than was possible without PCL.

Furthermore, Figure 4 tells us that without PCL the slower (less capable?) subjects produce solutions of much lower quality, whereas with PCL the quality is largely independent of the time required. This suggests that if a programmer is not able to solve a task, PCL may help recognizing that fact; the solution will be bad, but the time is not longer.

The WUSTL results for *And/Or-tree* indicate that the PCL group is faster, as shown in the lower part of Table 3. We find no clear difference in the number of points (line 5) nor in the number of completely correct solutions (line 6) — presumably because this group not only designed but also could implement and test their solution — but we find a large advantage in the time required for the group with PCL (line 7). With a confidence of 90 percent, having PCL saves between zero and 43 percent of the maintenance time for this task. Due to the small number of subjects we cannot evaluate statistically the results of just the completely correct solutions alone, but the trend remains the same.

As for learning effect, the UKA subjects were on average significantly faster (but did not obtain more points) in their second task. The size of the effect is independent of the presence or absence of PCL. This data is not shown in the table. Differences due to learning are compensated by the counterbalanced experiment design and are thus not relevant for the interpretation of the results. The learning effect could not be assessed for WUSTL, because the group sizes were too small. The shorter time for the second task could also be explained by a pressure to finish that builds as soon as the first few subjects have finished and left the room. However, such an effect would also result in reduced quality of the solutions, which we have not found, and is therefore unlikely.

Summing up, the *And/Or-tree* results suggest that for non-trivial pattern-relevant maintenance tasks the presence of PCL may save time and/or help avoid mistakes.

4.2 Results for *Phonebook*

The results are summarized in Table 4 and Figure 3. The *Phonebook* results of UKA show essentially no difference in the total number of points for the pattern-relevant subtasks (line 8), or the number of solutions that were completely correct (line 9). Although almost half of the participants made at least one mistake, the rather high relevant point values (over 16 out of a possible 18) indicate that the task was simple for these subjects.

Despite the simplicity, however, the group with PCL managed to solve the task faster than the group without PCL, as shown in line 10. The advantage can also be quantified: it has an expected

Table 4: *Results for the Phonebook task. The* WUSTL *results had to be discarded, because the subjects were not sufficiently qualified for this task; only one subject per group came up with a correct solution.*

	Variable	mean with PCL D^+	mean w/o PCL D^-	means difference (90% confid.) I	significance p
UKA, program *Phonebook*:					
8	relevant points	16.1	16.3	$-8.0\% \ldots + 4.0\%$	0.35
9	#corr. solutions	17 of 36	15 of 38		0.64
10	time (minutes)	51.5	57.9	$-22\% \ldots + 0.3\%$	0.055

size somewhere between zero and 22 percent (with 90% confidence); see also Figure 5. To ensure that the time difference is not some weird artifact related to different behavior of the subjects making serious mistakes, we may also compare only the times of those subjects that made no or almost no mistake in their solution. We find that the time difference still points in the same direction (not shown in the table). The learning effect was similar to that described for *And/Or-tree* above.

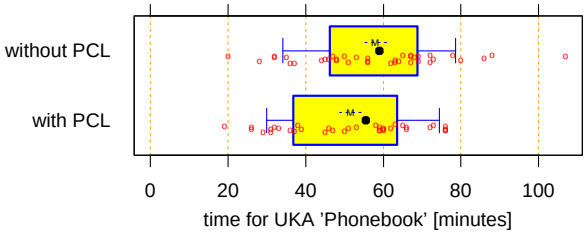


Figure 5: *Work time distribution for UKA Phonebook task. The fat dot is the median; the box indicates the 25% and 75% quantiles; the whiskers indicate the 10% and 90% quantiles; the M and dashed line indicate the mean plus/minus one standard error of the mean.*

Unfortunately, the WUSTL results for this task are meaningless, since there is only a single correct solution in each group. Thus, it makes little sense to compare the results. Our postmortem analysis found that for the given subjects the task was too difficult 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 in the program that they could imitate (in contrast to the UKA *Phonebook* program). Our results suggest that under such circumstances, PCL might be worthless.

Summing up, the *Phonebook* results suggest that for simple pattern-relevant maintenance tasks the presence of PCL may save time.

5 Interpretation and conclusions

In summary we find that our results support both of the hypotheses introduced in Section 2.1. Each of our results either supports hypothesis H1 (pattern-relevant maintenance tasks will be completed faster if PCL is added) or hypothesis H2 (fewer errors will be committed in pattern-relevant maintenance tasks if PCL is added). For UKA *Phonebook* and for WUSTL *And/Or-tree*, the PCL group was faster than the non-PCL group. We found the size of the effect (0 to 40 percent speedup) to be considerable, although this may not generalize to other cases. For UKA *And/Or-tree*, the PCL group produced solutions with fewer mistakes than the non-PCL group. In particular, twice as many subjects arrived at a completely correct solution. None of the three results supports both hypotheses at once, but there is no evidence for the opposite of the hypotheses, either.

Note that the design of these experiments was rather conservative; our design decisions biased them towards *not* showing any effects from adding PCL, in particular:

1. The programs were rather small, so even without PCL the subjects could achieve good program understanding within a reasonable time. In real software projects, PCL might be more helpful if the fraction of program understanding effort that PCL can save grows with the size of the program.
2. 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 PCL; see Section 3.4. In contrast, many programs in the real world lack design documentation. PCL is probably an efficient form of design documentation, as it is rather compact.

Given these circumstances, performance advantages

through PCL may often be even more pronounced in real situations than in our experiments. In any case, even a few percent reduction in maintenance effort outweigh the additional development cost for inserting PCL into source programs, at least if PCL is inserted right during development.

We conclude that depending on the particular program, maintenance task, pattern knowledge, and personnel, PCL in a program may considerably reduce the time required for a program change or may help improve the quality of the change. We therefore recommend that design patterns always be documented explicitly in program source code.

5.1 Why is PCL beneficial?

The analysis of design information units (see Section 3.4) suggests two answers to this question.

First, we often found the information provided by PCL much clearer than that in the other comments, which was sometimes vague, ambiguous, or spread out (delocalized). This was to be expected, because a clear and compact terminology is considered one of the primary advantages of patterns.

Second, PCL repeats some design information units more often than normal comments do, in particular with respect to mentioning the presence of a pattern; see Table 2. This type of repetition has two advantages: it reduces the concentration required to capture the relation between the pattern parts and it makes readers aware of the pattern even if they do not pick an appropriate exploration path through the program text.

This property makes PCL similar to the rather voluminous documentation that Soloway et al. [21] suggest for structured programs in order to overcome problems from *delocalized plans* — design decisions whose consequences are spread out over multiple parts of a program. Wilde et al. [27] argue that delocalized plans are frequent in object-oriented programs due to inheritance, large numbers of small methods, and dynamic binding. It seems that design patterns are a good means for understanding many such delocalized plans during maintenance, provided that PCL provides very powerful beacons that point out where patterns were used – each explaining a significant fraction of the overall program structure.

Likewise, the effectiveness of PCL can be explained using the theory of Brooks [2]: PCL can be considered to provide a bridge between the knowledge domains "design idea" and "understanding a single method or class", because design ideas are explained by mentioning a design pattern (which PCL does) and PCL is attached to individual class or method heads.

5.2 Further work

We 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 or when correct results are enforced by an acceptance test? 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 largely uncommented and how, in general, does PCL interact with other documentation? Sixth, what fraction of the effect is explained simply by the fact that PCL is a form of structured (rather than arbitrary style) comments? To what degree is it relevant how PCL is phrased or formatted? Finally, is PCL also helpful during code 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 PCL may slow down architectural erosion and drift (as described by [16]), 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 carry out the experiment at Washington University. We also thank all our experimental subjects for making this project possible. Thanks also to several critical reviewers of previous versions of this paper, in particular for pointing out that a methodological contribution was waiting to be made.

References

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

[2] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proc. 3rd Intl. Conf. on Software Engineering*, pages 196–201. IEEE CS Press, 1978.

[3] Ruven Brooks. Towards a theory of the comprehension of computer programs. *Intl. J. of Man-Machine Studies*, 18(6):543–554, June 1983.

[4] 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.

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

[6] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*, Monticello, IL, 1995. Addison-Wesley.

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

[8] EIR. Web pages of the Karlsruhe Empirical Informatics Research group. http://wwwipd.ira.uka.de/EIR/.

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

[10] Edward M. Gellenbeck and Curtis R. Cook. An investigation of procedure and variable names as beacons during program comprehension. In *[12]*, pages 65–81, 1991.

[11] 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. ftp.ira.uka.de.

[12] Jürgen Koenemann-Belliveau, Thomas G. Mohrer, and Scott P. Robertson, editors. *Empirical Studies of Programmers: Fourth Workshop*, New Brunswick, NJ, December 1991. Ablex Publishing Corp.

[13] Stan Letovsky. Cognitive processes in program comprehension. In *[22]*, pages 58–79, 1986.

[14] Robert Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design 3*, Monticello, IL, 1997. Addison-Wesley.

[15] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[16] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[17] 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.

[18] 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. http://www.cs.wustl.edu/cs/cs/publications.html.

[19] Mary Shaw and David Garlan. *Software Architecture — Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1996.

[20] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238, 1979.

[21] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[22] Elliot Soloway and Sitharama Iyengar, editors. *Empirical Studies of Programmers: First Workshop on Empirical Studies of Programmers (Washington, D.C.).* Ablex Publishing Corp., Norwood, NJ, June 1986.

[23] Barbara Unger and Walter F. Tichy. Do design patterns improve communication? An experiment with pair design. In George Stark, editor, *WESS: International Workshop on Empirical Studies of Software Maintenance*, http://members.aol.com/GEShome/wess2000/unger-tichy.pdf, October 2000.

[24] William N. Venables and Brian D. Ripley. *Modern Applied Statistics with S-PLUS*. Springer-Verlag, 2nd edition, 1997.

[25] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design 2*, Monticello, IL, 1996. Addison-Wesley.

[26] Anneliese von Mayrhauser and Stephen Lang. A coding scheme to support systematic analysis of software comprehension. *IEEE Trans. on Software Engineering*, 25(4):526–540, July 1999.

[27] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, January 1993.

[28] *Proc. 5th Intl. Workshop on Program Comprehension*, Dearborn, MI, March 28–30, 1997. IEEE Computer Society.

[29] *Proc. 6th Intl. Workshop on Program Comprehension*, Ischia, Italy, June 24–26, 1998. IEEE Computer Society.

[30] *Proc. 7th Intl. Workshop on Program Comprehension*, Pittsburgh, PA, May 5–7, 1999. IEEE Computer Society.

[31] *Proc. 8th Intl. Workshop on Program Comprehension*, Limerick, Ireland, June 10–11, 2000. IEEE Computer Society.