WASHINGTON · UNIVERSITY · IN · ST · LOUIS

Replication of the first controlled experiment on the usefulness of design patterns: Detailed description and evaluation

Lutz Prechelt, Barbara Unger, Douglas C. Schmidt
(prechelt | unger@ira.uka.de, schmidt@cs.wustl.edu)
Department of Computer Science
Washington University, St. Louis, MO 63130-4899

Technical Report wucs-97-34
December 2, 1997
http://www.cs.wustl.edu/cs/techreports/1997/

Abstract

Advocates of software design patterns claim that using design patterns improves communication between software developers. The controlled experiment that we describe in this report tests the hypotheses that software maintainers of well-structured, well-documented software containing design patterns can make changes (1) faster and (2) with less errors if the use of patterns is explicitly documented in the software.

The experiment was performed with 22 participants of a university course on C++ and design patterns; it is similar to a previous experiment performed in Karlsruhe.

For one of the two experiment tasks the experiment finds that both hypotheses appear to be true. For the other task the results are inconclusive, presumably because the task was too difficult for the given experience level of the subjects.

Contents

This is a one line proof
. . . if we start sufficiently far to the left
Anonymous math lecturer

Chapter 1

Introduction

The present report is the definitive and detailed description and evaluation of a controlled experiment on the influence of design pattern documentation on the maintainability of object-oriented programs.

In the first chapter we will first discuss the general topic of the experiment (design patterns), then give a broad overview of the purpose and setup of the experiment. We then describe the differences to the previous Karlsruhe pattern experiment and explain how to read this report.

Chapter 2 describes the preparation, setup, and execution of the experiment, relying heavily on the original experiment materials as printed in the appendices. It also discusses threats to the internal and external validity of the experiment.

Chapter 3 presents and interprets in detail the results obtained in the experiment and Chapter 4 presents conclusions and relates the results to our previous ones. The appendices contain the handouts used in the experiment: questionnaire and program source code.

1.1 Design Patterns

Several years ago, a group of researchers and practitioners from the object-oriented design and programming community began to collect descriptions of proven solutions to recurring problems in object-oriented design. Such *design patterns* package expert knowledge into a form that can be reused frequently and easily. Rapidly, these collections have become a promising development for making design a more sound activity in software engineering. A documentation format was developed and today a design pattern is a packaged description of a common software design problem, its context, appropriate terminology, one or several solutions, and their advantages, constraints, and other properties.

According to practitioners [1, 6], there are several advantages of design patterns:

- Less experienced designers can produce better designs with patterns.
- Design patterns encourage recording and reusing best practices even for experienced designers.
- Design patterns can improve communication, both between designers and from designers to maintainers, by defining a common design terminology.

The first large, orderly collection of design patterns was presented in 1995 as a book by Gamma, Helms, Johnson, and Vlissides [6], often called the "Gang of Four (GoF)." The GoF book enjoyed great success and caused significant interest in design patterns.

Currently the number of design patterns reported in the literature is rapidly increasing and there are several conferences on the topic [10]. Some newly documented design patterns are appealing, though many are variations of previously documented patterns. The idea of design patterns is also extended in other directions: Groups of patterns are presented as so-called "pattern languages," pattern taxonomies are suggested, patterns on higher levels of abstraction (architectural patterns) or lower levels of abstraction (idioms) are collected, formalizations are sought, tools are built for discovering new patterns or for recovering known patterns from existing software etc. [2].

As often occurs in software engineering, the activities related to patterns are built on subjective beliefs, rather than empirically validated knowledge, that the developments are useful. Currently, these beliefs are grounded in intuitive judgement or anecdotal evidence reported by practitioners from the pattern community [1].

We believe that systematic tests of the purported advantages of patterns should be conducted to understand the mechanisms: *e.g.*, whether, why, when, and to what extent these advantages exist. Such tests will also help avoid expensive developments in useless or less fruitful directions. The tests may come in the form of case studies, larger field studies, or controlled experiments; a combination of all three will be required before we really understand design patterns. In this report we present a replication (with some variations) of the first controlled experiment for testing some aspect of the usefulness of patterns.

1.2 Experiment Overview

One of the advantages that design patterns are assumed to have is improved communication: They provide a powerful and well-defined terminology that speeds up communication and minimizes misunderstandings of software design and architecture.

As mentioned above, such communication can occur during design time, implementation time, or program maintenance. Our experiment considers the latter situation: Do design patterns help the maintainer to understand the design so that s/he can make the desired changes faster, more correctly, or with less negative impact on the structure of the software?

More precisely, our experiment investigates the following: Assume the maintainer knows what design patterns are and how they are used. Furthermore, assume that the program in question was explicitly designed using patterns known by the maintainers. Now the question is:

> Given a thorough program documentation, does it help the maintainer if the design patterns in the program are documented *explicitly*, as opposed to a documentation that merely describes the resulting structure as it is?

We investigated this question in the following manner: Subjects in the experiment received the same program (C++ source code) and the same change requests for that program; they had to outline how the changes should best be done. The program was documented in detail but the subjects in subgroup A received no explicit information about design patterns in the program, whereas subgroup B received the equivalent program with the design patterns explicitly marked and named in a small amount of additional documentation embedded in the source code. We investigated whether and how the performance[1] of group A was different from group B.

The experiment was performed with 22 student subjects in a single session ranging from 2 to 4 hours. The tasks were based on two programs of 8 to 10 printed pages length; solutions had to be implemented on each subject's Unix workstation.

1.3 Differences to Karlsruhe Experiment

The experiment described in the present report is based on another experiment that was conducted at the University of Karlsruhe in January 1997 [8, 9]. These are the most important differences of the present experiment compared to the former one (readers not familiar with the Karlsruhe experiment may want to skip this section):

1. The solutions were produced directly in the computer and could be compiled. The Karlsruhe experiment was conducted on paper only.
2. Some of the subtasks were left out due to time constraints.
3. The programs were written in C++ instead of Java. Except for the changes noted below, however, the programs were functionally and architecturally equivalent to the Java programs of the Karlsruhe experiment.
4. The *Tuple* program was not a GUI program, as most participants had no sufficient prior knowledge of any single GUI library (see Figure 2.5 on page 12).
5. In the Karlsruhe experiment, there was always a class present in the given program that was structurally rather close to the new class the subjects had to introduce. This class could be used as a model to simplify the task: By mimicking the model class, the task could be solved without real program understanding. For the *Tuple* task, these model classes were left out of the programs in the new experiment.
6. The course at whose end the experiment was performed had more design pattern content compared to the Karlsruhe course. Therefore the participants knew more different design patterns than the Karlsruhe participants.
7. Here we had only 22 participants instead of 74 and all subjects were undergraduate students.

The present report matches [8] as closely as possible with respect to its structure and style of presentation.

1.4 Related Work

Except for the Karlsruhe experiment mentioned above, no scientific investigation of the assumptions underlying design patterns has yet been published as far as we know. The only other reports available are experience reports and anecdotal evidence from researchers and practioners in the design pattern community [1].

[1] measured by the correctness of the solution and the time taken for constructing it.

1.5 How to Use This Report

This report is meant to provide a most detailed documentation of the experiment and its results. That means that it should not be read sequentially from front to back, but instead **the appendix needs to be consulted when reading the text**: The main text does not try to describe the tasks or questionnaires in much detail but instead relies on the original experiment materials (questionnaires, task sheets, program sources) that are printed in the appendix.

Chapter 2

Description of the experiment

2.1 Experiment Idea and Hypotheses

Improved communication is one of the purported advantages of design patterns. Such communication occurs within or between different groups of people (designers, implementors, maintainers) and in different modes (either interactively in real time or one-way via documents). Our experiment attempts to investigate the situation of implementor-to-maintainer communication via the source code document. This is an important question, because maintenance is known to be a large cost factor in software engineering. In particular, the decay of software structure (architectural erosion [7]) during maintenance arises from the lack of design understanding.

In our experiment we investigate the following question:

> Assume a program was built using design patterns, was thoroughly documented, and now needs to be maintained. Is it useful to have a small amount of additional documentation that explicitly describes the design patterns used in the program? In particular, can software maintenance then be performed quicker, safer, or better?

The basic idea of the experiment is the following: Produce a program using design patterns and come up with a number of change requests for that program. Document the program thoroughly, but without explicit description of the design patterns used in the program. Give this program to one group of experimental subjects and let them do the changes. Give the same program to another (equivalent) group of subjects, but insert a small amount of additional documentation (called the *pattern documentation*, PD) into the program source code that describes the use of design patterns using standard design pattern terminology. Make the amount of PD so small and the rest of the documentation so complete that the PD does not provide information about program structure that is not present otherwise; PD should only add another view.

The expected outcome is that on average the group without PD in the program will take longer to finish or produce solutions with worse structure or will have more errors in their solutions. More specifically, we investigate the following two hypotheses:

Consider the following type of maintenance task, which we call *pattern-relevant task*: Given a program that uses design patterns and that is well commented we call a maintenance task *pattern-relevant* if (1) it touches (is influenced by) one or several uses of design patterns in the program and (2) performing the task requires understanding at least a substantial part of the software structure embedded in these design patterns. A task is only *partially pattern-relevant* if understanding is one option for solving the task but there are also other ways of solving it.

Hypothesis H1: Pattern-relevant maintenance tasks will on average be completed quicker if PD is given in a program than if PD is not given in the otherwise same program.

Hypothesis H2: Likewise, fewer errors will be committed on average in pattern-relevant maintenance tasks if PD is given in a program than if no PD is given.

Another interesting hypothesis is that correct (i.e, functional) solutions of the tasks will, on average, better maintain the intended design structure of the program, i.e., avoid structural decay. This hypothesis is somewhat difficult to test objectively, since it is often a matter of taste and expectations what the best design would be. Therefore we did not formally test this hypothesis in the experiment.

2.2 Experiment Design

To balance differences of ability between the groups and to get more data, the actual experiment used two different programs (*Element* and *Tuple*, see Appendix B on page 54) and each subject worked on both (one with PD and one without). We measured the time taken by each individual subject and judged the solutions that they delivered.

The independent variable in this experiment is 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 (22 non-empty lines of PD added to the 498 line program *Element*, 10 non-empty lines added to the 448 line program *Tuple*), the other had no such additional documentation.

The dependent variables are the time required to complete all tasks given for each program, the degree of correctness of the solutions for each task, the types of errors found in a solution, and subjective information from a postmortem questionnaire.

We also administered two short questionnaires immediately before the actual experiment: One for gathering statistical information about our subjects and another for testing their knowledge of design patterns.

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 [3]; see Table 2.1.

Furthermore, we also balanced the four resulting groups for expected subject ability, measured by the percentage of points each subject received in the lab course, using stratified random sampling: We grouped the subjects into blocks of similar percentages and evenly distributed the subjects from each block into the four experimental groups in a random fashion. 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. All the subjects knew was that the experiment was about design patterns and all the subjects had studied the GoF patterns during the semester.

2.3 Preparation: The CS242 Course

In order to have enough subjects with sufficient ability and comparable background, we gave a course at Washington University, St. Louis to “breed” our subjects. This CS242 course provided intensive focus on practical aspects of designing, implementing, and debugging object-oriented software. Topics covered included reuse of

	first with PD then w/o PD	first w/o PD then with PD
first Element, then Tuple	E^+T^-	E^-T^+
initial no. of subjects	6	5
number of data points, *Tuple*	4	3
number of data points, *Element*	4	4
first Tuple, then Element	T^+E^-	T^-E^+
initial no. of subjects	6	5
number of data points, *Tuple*	3	3
number of data points, *Element*	4	4

Table 2.1: The four experiment groups and their size. The number of data points is one for each subject, except for those subjects that did not complete the respective task, but dropped out of the experiment instead. (E^+T^- stands for "first perform Element with PD, then perform Tuple without PD" etc.)

design patterns and software architectures, as well as developing, documenting, and testing representative applications using object-oriented frameworks and C++. Design and implementation based on design patterns and frameworks were central themes to enable the construction of reusable, extensible, efficient, and maintainable software.

CS242 is a popular class at Washington University. The Spring '97 offering attracted 23 highly motivated students. The course lasted 15 weeks, with two 90-minute lectures per week and one programming assignment due every two weeks. The exercises were submitted electronically and were graded by the instructor in one-on-one sessions with the students.

There were five programming assignments in the course, each of which introduced various design patterns to the students. Assignment 1 involved developing dynamic array and stack abstract data types. This exercise focused on the Wrapper and Adapter patterns. Assignment 2 was a program that checked for balanced delimiters. This exercise introduced the Template Method and Bridge patterns. Assignment 3 was a system sort utility using optimized quicksort. This assignment focused on the Singleton, Bridge, Facade, and Strategy patterns. Assignment 4 was a validation program for the system sort program of assignment 3. This assignment also focused on the Strategy and Factory Method patterns. The final assignment was on operator precedence parsing and binary trees. This program illustrated the Visitor, Strategy, Factory Method, Interpreter, and Builder patterns.

Thus, our participants practiced 11 design patterns: Wrapper, Adapter, Template Method, Bridge, Singleton, Facade, Strategy, Factory Method, Visitor, Interpreter, and Builder. Each of these was shortly introduced in the lecture and further motivated and explained in the assignment descriptions. The latter made quite precise prescriptions where and how to apply the design patterns, so actual practice with them was ensured. In addition, all course participants attended a series of lectures during the same semester where they presented and learned about all 23 design patterns in the GoF book in a conceptual fashion without hands-on experience.

Unfortunately, only one of the patterns relevant for the experiment (namely the Template Method) was actually exercised in an assignment by the students. This is because the course and the experiment were designed independently.

On the other hand, this means the experiment will test effects of PD for patterns with which the programmer is familiar only in a theoretical fashion — a situation that will not be uncommon in practice.

2.4 Experiment Format and Conduct

Participating in the experiment and showing reasonable performance was required to receive course credit for the preparation course. All participants needed that credit.

We carried out the experiment in May 1997 on a Tuesday afternoon, 15:30 to 19:00 hours. The experiment was conducted as a laboratory exam in two workstation pool rooms. The questionnaires and task descriptions were administered on paper. The source programs were available as files. The solutions were constructed on the computer with the help of a compiler; they had to be delivered in files.

The documents were handed out in five parts as described in Appendix A on page 39. The exam was announced as taking two hours, but each subject could work in his or her personal pace and was given as much time as s/he wanted. The materials for each task were handed out and collected separately, one after the other. This way we could collect reliable time information about each task for each subject individually. For all further details see the actual documents as used in the experiment; they should be self-explanatory and are printed in the appendices starting on page 39.

2.5 Experimental Subjects

2.5.1 Education

22 subjects participated in our experiment. All of them were Computer Science students (21 male, 1 female) On average, they were in their 6th term at the university; see Figure 2.1.

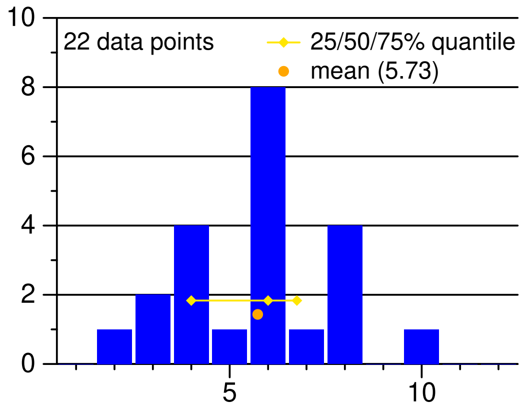


Figure 2.1: Distribution of term numbers of experimental subjects.

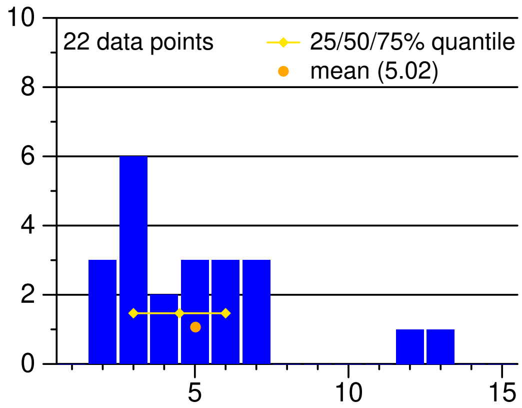


Figure 2.2: Distribution of years of programming experience of subjects.

2.5.2 Programming experience

These students had an average of 5 years of programming experience (Figure 2.2 above), 91% of them had written more or much more than 3000 Lines of Code (LOC) in their lifes (Figure 2.3). 73% had significant practical experience with object-oriented programming (Figure 2.4) and 50% had significant practical experience in programming graphical user interfaces (Figure 2.5). Our subjects had practice with an average of 4

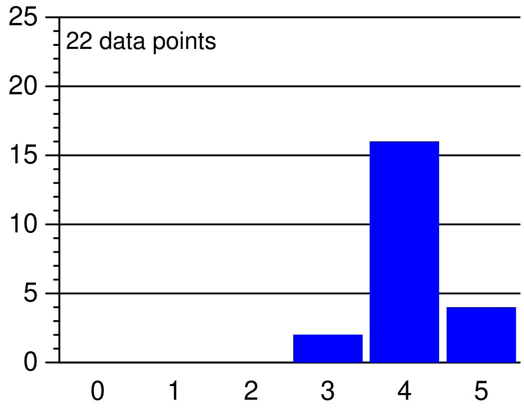


Figure 2.3: Distribution of previous programming knowledge and experience: 0=no knowledge, 1=only theoretical knowledge, 2=less than 300 LOC written, 3=less than 3000, 4=less than 30000, 5=more than 30000. The same encoding is used in the other two programming experience histograms.

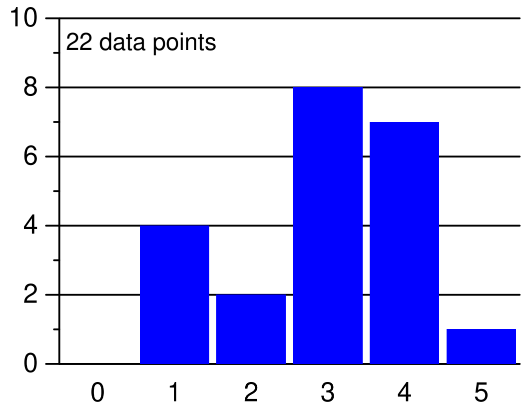


Figure 2.4: Distribution of previous experience in object-oriented programming.

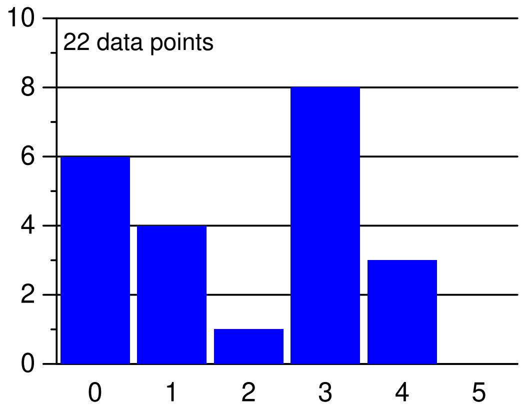


Figure 2.5: Distribution of previous experience programming graphical user interfaces (GUI).

different programming languages (Figure 2.6 below). The largest program ever written by our subjects had an average size of 2557 LOC (median: 2000 LOC) and 2.7 person months (median: 1 person month); see Figures 2.7 and 2.8 below. 18% of the subjects had also previously participated in a team software project and contributed an average maximum of 7700 LOC (median: 2800 LOC) and 5.6 person months (median: 5 person months) to the total project size of on average 50133 LOC (median: 50000 LOC) and 62 person months (median: 27 person months); see Figures 2.9 to 2.12 on page 14.

2.5.3 Knowledge of Design Patterns

Before the actual experiment started, we tried to learn about our subjects' knowledge of design patterns in two ways. First we asked "estimate subjectively how well you understand the following design patterns." Answers

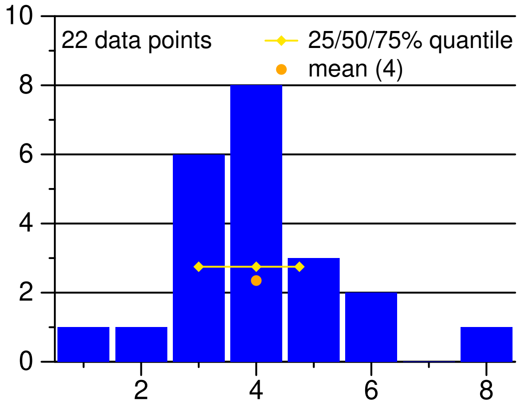


Figure 2.6: Distribution of number of programming languages previously used.

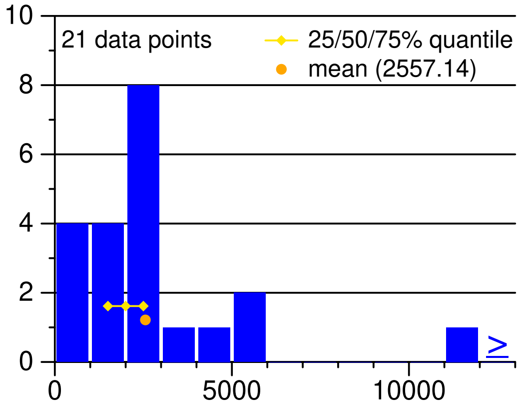


Figure 2.7: Distribution of size (in LOC) of largest program ever written alone.

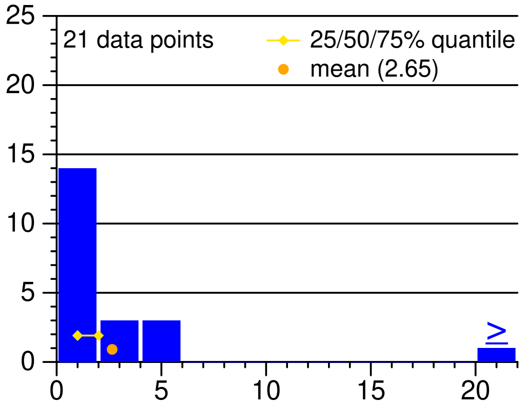


Figure 2.8: Distribution of size (in person months) of largest program ever written alone.

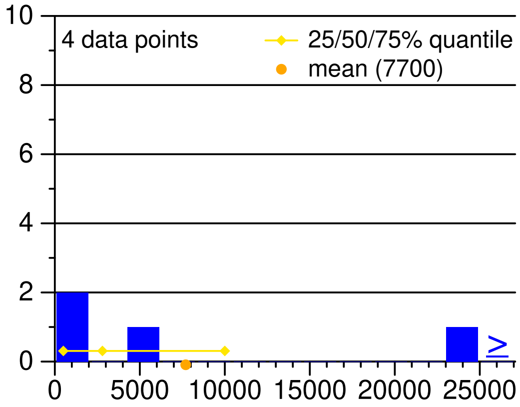


Figure 2.9: Distribution of size (in LOC) of subject's contribution to his/her largest team software project.

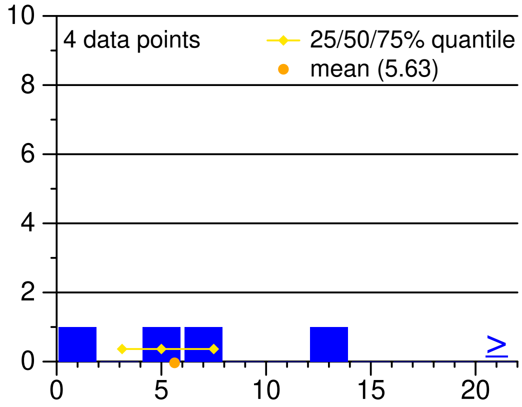


Figure 2.10: Distribution of size (in person months) of subject's contribution to his/her largest team software project.

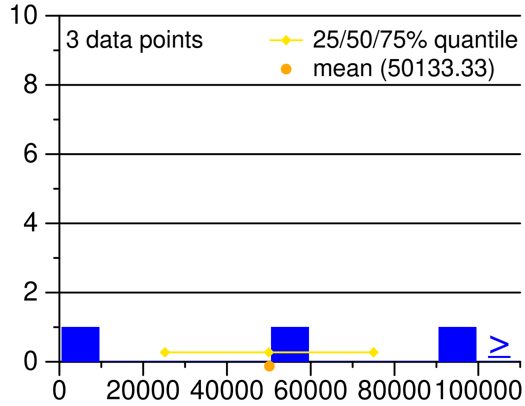


Figure 2.11: Distribution of size (in LOC) of largest team software project of subject.

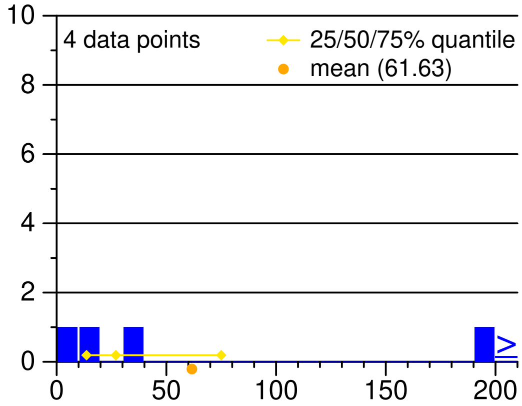


Figure 2.12: Distribution of size (in person months) of largest team software project of subject.

were on a qualitative five point scale from 1:"I understand the pattern very well" to 5:"I do not understand it at all".[1]

Our subjects claimed for themselves rather different knowledge of different design patterns: average grades range from 1.43 (quite good knowledge) down to 3.67 (very little knowledge); see Figure 2.13 below. For the four patterns relevant to our experiment, only knowledge of the Template Method (average grade 2.05) was claimed to be good. For each of the other three, a majority of participants claims "rough understanding" at best and the averages range from 2.63 to 2.71, with considerable spread in each of the distributions. In the Karlsruhe experiment the four relevant patterns all received average grades in the range 1.82 to 2.16.

This indicates a possible problem with the population used in the new experiment. Except for Template Method, the patterns relevant for the experiment were only discussed shortly towards the end of the course. Many participants may not understand the patterns well enough for taking advantage of PD.

In the second questionnaire, we conducted an actual test of pattern knowledge; see Appendix A on page 39 for its exact form.

The first question of the test asked in which of 8 given design patterns which of 6 given operations usually occur. Only the 6 positive ('yes') answers needed to be given by placing a mark in the table, all other fields of the table could be left empty. This question requires active or passive knowledge or thorough understanding of the patterns for identifying the 'yes' answers as well as active knowledge or thorough understanding for avoiding the wrong ones. We summarize the answers to all of the 48 subquestions in a single number of points. The rules for assigning these points are shown in Table 2.2 on page 16. The best possible result was 12 points, the worst possible was -53 points. The actual range obtained by our subjects was from -10 to 11 points. See Figure 2.14 below for the point distribution. We find that half of the subjects obtained 5 or more of the 12 points, which is a reasonable level of pattern knowledge given the large number of possibilities for scoring negative points. One quarter of the subjects obtained 8 or more points. Roughly one quarter of the subjects obtained a negative sum of points and must be considered having low pattern knowledge. These results are similar to the Karlsruhe experiment.

The second question asked "what is the alternative to the introduction of a Visitor pattern?". This was directly relevant for the Element task later in the experiment. The correct answer (A) is adding a method to each class

[1] We treat this scale as an interval scale in the following.

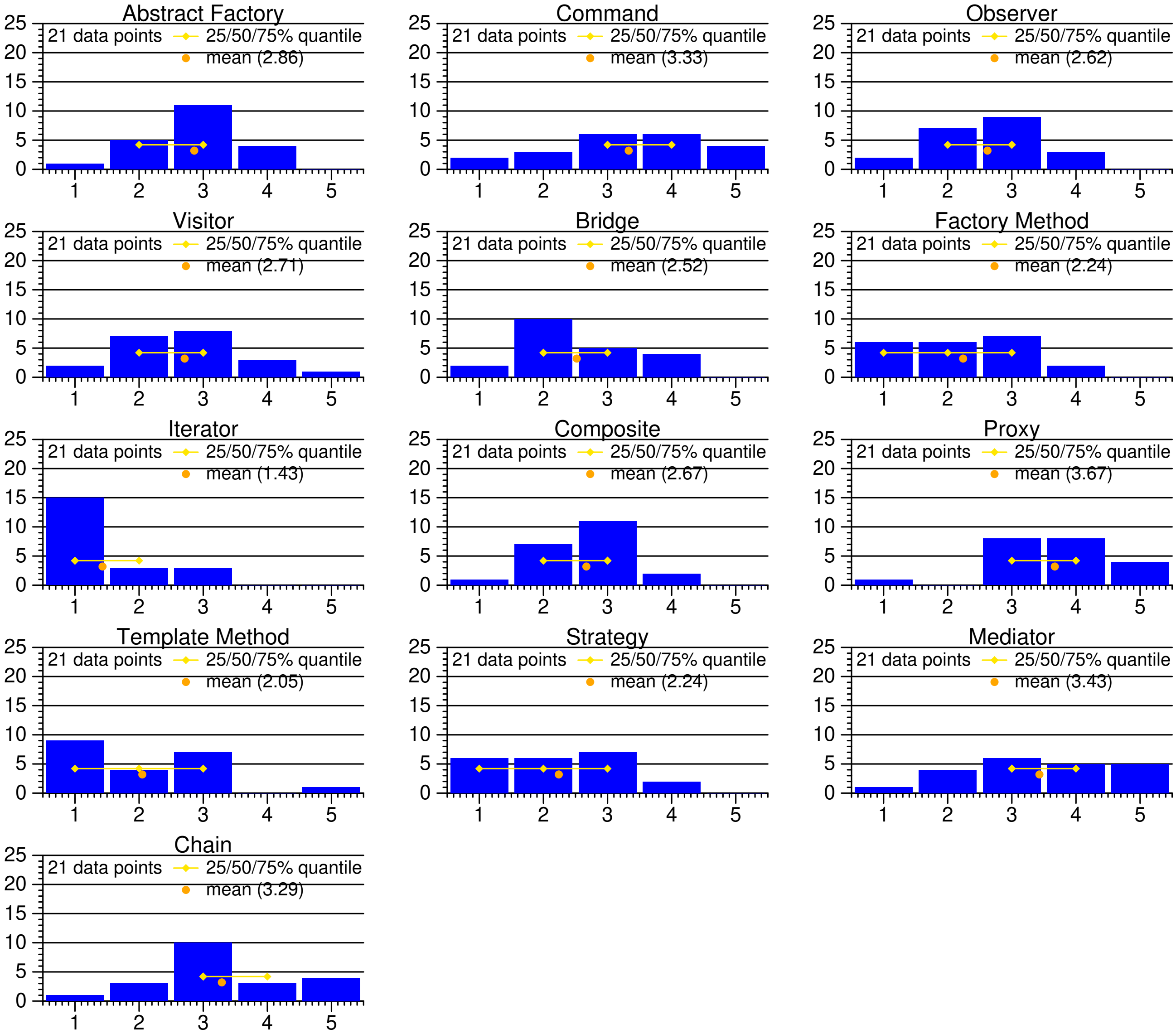


Figure 2.13: Subjective estimation of pattern knowledge. 1=understand very well, 2=understand well, 3=understand roughly, 4=begin to understand, 5=do not understand.

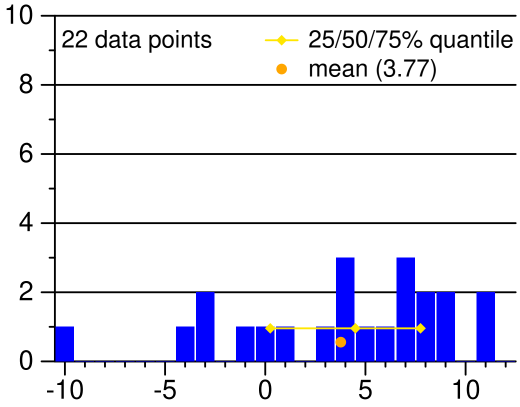


Figure 2.14: Distribution of points obtained in first question of pattern knowledge test.

Operation	Com mand	Obser ver	Visi tor	Com posite	Tem plate Meth.	Stra tegy	Medi ator	Chain of Resp.
accept()	-1	-1	**2**	-2	-2	-1	-1	0
register()	-2	**2**	-2	-1	-2	-1	-2	0
execute()	**2**	-2	-1	-2	0	0	-1	0
add()	-2	0	-2	**2**	-2	-1	-1	0
notify()	-1	**2**	-1	-2	-2	-1	-1	-1
update()	-2	**2**	-2	-2	-2	-1	-2	-1

Table 2.2: Points given for a 'yes' mark for each of the fields in the table of first question of pattern knowledge test. No 'no' marks were required. Correct answers give 2 points each (printed in boldface), wrong answers give -1 or -2 points, depending on the degree of absurdity. Some answers would be arguable and give 0 points.

to be visited; we counted 3 points in this case. Other answers such as using an Iterator (G) have at least a little truth, but are not universal and therefore counted only 1 point. Sometimes the correct answer was inflicted with additional suggestions that were wrong, e.g. to use subclasses (M), or was stated rather vaguely. We counted 1 to 3 points in these cases. Figure 2.15 shows the point distribution for this question. We see that 68 percent

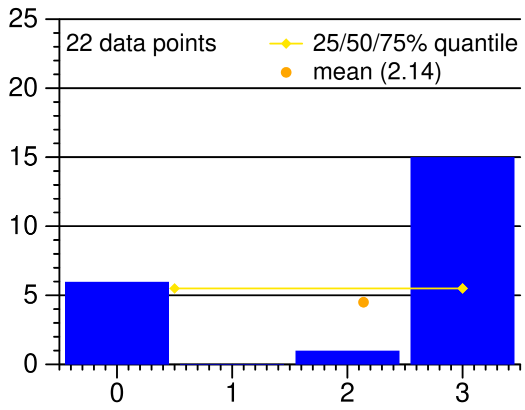


Figure 2.15: Distribution of points for second question of pattern knowledge test: What is the alternative to introducing a Visitor?

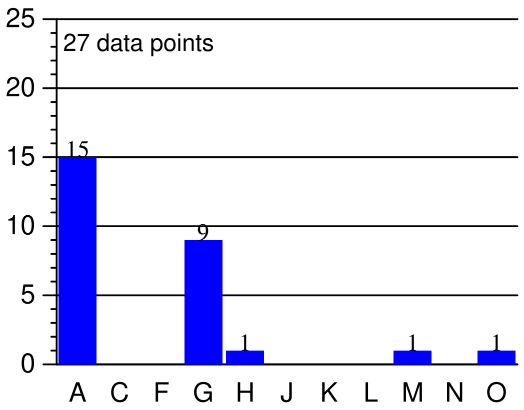


Figure 2.16: Frequency of different answers for this question: A:introduce new method, C:Strategy, F:Mediator, G:Iterator, H:Observer, J:Adapter, K:Bridge, L:Template Method, M:use subclasses, N:Chain of Responsibility, O:call available methods.

of all subjects gave the right answer. 5 of these subjects suggested additional methods that were wrong or less good, but they still received all 3 points.

As for the frequency of the most common suggestions, see Figure 2.16. Only one subject made a nonsensical suggestion of using an Observer(H); five others also only had the wrong solution Iterator (G). As in the first question, however, overall pattern knowledge seems acceptable for a majority of our subjects.

We did not evaluate at all the third and fourth question of this test. The third question resulted in answers that were often difficult to judge. Evaluating this question would have introduced too much subjective bias. The fourth question was of only minor interest for the experiment; it was mainly meant to distract the participants from the patterns relevant for the experiment.

2.6 Tasks

This section will shortly describe the tasks and will explain why we chose them. You can find the original task descriptions in Appendix A and the corresponding program listings in Appendix B.

2.6.1 Constraints

The tasks used in our experiment had to obey the following constraints:

1. The experiment had to be carried out in a single time interval, as opposed to software development over several days or weeks. On average the tasks should consume less than one hour each, because the exam was announced to be two hours long (with open end, though).
2. The application domains of the programs had to be well understandable by all subjects. Therefore we could only use domains that were either known from the course or were so simple that they could be explained in a few words and understood readily.
3. The tasks should preferably only employ patterns that had been practiced in the course.

2.6.2 How Constraints Were Handled

We handled these constraints as follows. Constraint 1 rules out large programs or complex change requests. Therefore, we used programs of a few hundred lines and straightforward tasks.

As the experiment was a replication of the Karlsruhe experiment, yet the underlying course did not perfectly match the given tasks, constraints 2 and 3 could only partially be obeyed: The course did not cover GUI programming, therefore the Observer task “Tuple” (see below), although simple enough in principle, was not very natural for the subjects. The Visitor task “Element” was better in this respect. Except for Template Method, the relevant patterns for both tasks had not been covered thoroughly in the course.

The subjects were orally told to write only comments instead of program code where they found the task to be too difficult; most subjects produced at least the class, attribute, and method declarations and left out only method bodies.

2.6.3 Task “Tuple”

The “Tuple” program reads, stores, and displays structured tuples of firstname, lastname, and phone number. The store/display part is organized according to the Observer pattern. One actual display class is implemented, generating a simple chronological list of entries. An abstract superclass is provided that can be used for implementing a variety of different display classes: The superclass contains a Template Method providing 3 slots for filling in arbitrary methods for selecting, sorting, and formatting tuples.

The actual subtasks require the following: Finding a particular spot in the program (subtask 1); introducing a new display class using the Template Method superclass (subtask 2); introducing another new display class, similar to the existing one (subtask 3). Please see the exact task descriptions of subtasks 2 and 3 in the appendix, as they are central for the experiment.

The program will infinitely go through the following loop: read another tuple from the keyboard, then for each existing observer redisplay its contents line by line in the appropriate order and format on standard output, one observer after the other, separated by headlines. This output style makes it more difficult to get an intuitive grasp on the Observers than in a GUI program where each Observer is a fixed separate screen entity. New observer objects will be created only at program start, before the loop is entered. For the initial program there is but one observer, for the final program there are three.

2.6.4 Task "Element"

The "Element" program contains a simple library for constructing AND/OR-trees of character strings. AND is interpreted as concatenation and OR is interpreted as alternation, so that a tree defines a set of alternative strings. The library has methods for constructing AND nodes, OR nodes, and leafs, for printing a tree in term form, and for iterating through a tree in order to compute the depth of the deepest AND node, deepest OR node, and overall deepest node.

The node classes are arranged as a Composite pattern (leaf nodes are the leafs, AND and OR nodes are the containers), the depth computation is realized in a separate class using a Visitor pattern.

The actual subtasks require the following: Finding a particular spot in the program (subtask 1); determining the expression u.variants().size() for computing the number of variants (subtask 2); introducing another Visitor class for efficiently computing the number of variants (subtask 3). Please see the exact task description of subtask 3 in the appendix, as it is central for the experiment.

Analog reasoning may be used to solve the task instead of actually understanding the program: Understanding what the depth computation does and what the variant counting computation must do, it is clear that the same class structure can be used. The depth computation, however, uses an auxiliary method iterate() for handling AND and OR nodes that is not useful for variant counting, which has to handle AND and OR differently. Therefore, we may expect that solutions found by analog reasoning often contain iterate(), while solutions found from deeper program understanding usually will not.

2.7 Internal Validity

There are two sources of threats to the interval validity of an experiment[2]: Insufficient control of relevant variables or inaccurate data gathering or processing.

As far as we can see, all relevant external variables have been appropriately controlled in this experiment. In particular, there is no bias in the random group sampling, the subjects seemed willing to perform as best as they could in both experimental conditions, environmental conditions were essentially the same for all subjects, and the counter-balanced experiment design partially controlled for accidental group differences, learning, and sequencing effects, if any. The counter-balancing control is only partial for the following reason: While the experiment itself was counter-balanced, the data analysis and result interpretation partly ignores the results for

[2]Definition from [3]: "*Internal validity* refers to the extent to which we can accurately state that the independent variable produced the observed effect."

the Tuple data (for reasons we will discuss in the results section). Therefore, some of the advantages of the counter-balanced experiment design are lost. However, the power of the experiment is still larger than that of an ordinary two-group, one-treatment experiment.

The dominant control problem is mortality: Some students gave up on one or both of the tasks, when they recognized it would (or might) be too difficult for them or take too long. Four students gave up on both tasks.

Fortunately, mortality happened almost exactly as often in the groups with PD as in those without PD. It is therefore safe to assume that the mortality does not bias the results, provided that we ignore entirely those data points for which the tasks were not completed — which is just what we choose to do.

We tried to minimize data gathering errors by exercising utmost care. Data processing was almost completely automated and we believe it to be accurate. Manual and automated consistency checks were applied for detecting various kinds of mistakes in data gathering or processing.

2.8 External Validity

There are three sources of differences between the experimental situation and real software maintenance situations that limit the generalizability (external validity) of the experiment: in real situations there are subjects with more experience, programs of different size or structure, and tasks of different kind or complexity.

The most frequent concern with controlled experiments using student subjects is that the results cannot be generalized to professional software engineers because the latter are so much more experienced. In the present case, this may either be an advantage or a disadvantage: Professional programmers may have less need for PD because of their experience but just as well they may also be able to exploit it more profitably than our student subjects.

Another obvious difference is program size. Compared to typical industrial size programs, the experiment programs are rather small. This will not invalidate any positive result of the experiment, though: It is highly plausible that with increasing program size, the benefits from PD, if any, can only *increase* as well, because PD provides program slicing information. For pattern-relevant tasks, PD points out which parts of a program are relevant and allows to ignore the rest; such information becomes the more useful the more source code can be ignored. It is impossible to predict what implications different application domains will have for our results, but it seems likely that at least domains that do not distort the design patterns' metaphors will behave similar to those of the experiment.

Finally, the kind of task and its complexity may be different. In the experiment, the kind of task was program additions by complete new classes and the complexity was rather low. The experiment does not really tell us about other kinds of task, but its results may be interpreted to indicate that tasks of higher complexity will benefit more from PD, see the discussion below.

Chapter 3

Experiment results and discussion

This chapter presents and interprets the results of the experiment. The first section explains the means of statistical analysis and result presentation that we use and explains why they were chosen. The second section presents the central results (subjects' objective performance) and the third section adds data from the post-mortem questionnaire for understanding some of the effects underlying the performance.

3.1 Statistical Methods

3.1.1 Inference

Most formal statistical reasoning used below is meant for comparing the means of pairs of distributions. Hardly any of these distributions are normal distributions: Most of them are discrete and coarse-grained, several of them have two peaks, and many are heavily skewed or even monotone. Therefore, statistical analysis must not use a parametric test such as the t-test that assumes a normal distribution. On the other hand, classical non-parametric tests, such as the Wilcoxon Rank-Sum Test, cannot perform inference for the mean but only for the median, which is less relevant for our purpose. Moreover, rank sum tests cannot provide confidence intervals.

In this report, we thus use resampling statistics (bootstrap) to compare the means of arbitrary distributions non-parametrically. The basic idea of resampling is considering the distribution of the sample to be the distribution of the underlying universe[1], as it is the best approximation of the actual distribution we have, unless we make assumptions. Instead of making assumptions and then using an analytical procedure for inference, resampling uses a computational procedure. In resampling one produces an arbitrary number of samples S_i from the given sample A by picking an arbitrary element of A at random each time. The chosen elements are not removed from A, so they can appear multiple times in the same S_i ("sampling with replacement"). This "re-sampling" produces arbitrary amounts of observations that directly simulate the universe from which A was taken. From these observations, a confidence interval for the target statistic (whatever it may be) can be computed directly.

In this report, the only resampling procedure used is the comparison of the means of two samples A and B (which may have the same or different size). To do this, we repeatedly draw resamples A_i and B_i from A and B, compute their means, and collect the set D of differences $d_i := \overline{A_i} - \overline{B_i}$. From the empirical distribution of D we directly read confidence limits for d and the significance of the difference.

[1]This principle is known as *plug-in* estimation.

Our resampling program for this purpose is written in Java using the package *resample* that is available from http://wwwipd.ira.uka.de/~prechelt/sw/. The core part of this program is roughly as follows:

```
/* a, b contains the sample A, B */
ResampleVector result = new ResampleVector();
ResampleVector resample_a,
               resample_b;
Double         d;
for (int i = 1; i <= 10000; i++) {  // number of resample trials
  resample_a = a.sample(a.size());  // take a resample from A
  resample_b = b.sample(b.size());  // take a resample from B
  // compute the difference of the resample means:
  d = resample_a.mean() - resample_b.mean();
  result.addElement(d);             // store the result
}
result.sort();
```

After this procedure, `result` contains an empirical distribution of 10000 differences, from which quantiles (for confidence limits) or inverse quantiles (at zero, for computing the significance of the difference) can be read: The 90% confidence interval for d ranges from `result.quantile(0.05)` to `result.quantile(0.95)` and the significance is `result.quantileWhere(0.0)` (or one minus that, depending of the sign of the difference). In the tables below, the confidence intervals are normalized and converted into percentages of `b.mean()`.

Of course resampling is no cure-all: If there is too little data, the confidence intervals will be imprecise. However, for our purposes, it works well: We have several dozen data points in each sample, one-dimensional distributions only and the underlying distributions either have only few distinct values or are quite smooth. Under such circumstances, enough data is available so that resampling produces reliable results. On the other hand our sample distributions have very different shapes and few of them are anything close to normal. In contrast to classical statistical methods, resampling avoids distributional assumptions and allows for using the same procedure in all cases.

A nice introduction into resampling for statistical laymen is by Simon [11]. Readers with deeper statistical knowledge may prefer the more mathematical yet highly understandable text of Efron and Tibshirani [4].

The only other statistical test used is the χ^2 test on a four field table for testing the significance of frequency differences of a binary attribute. The application is comparing the incidence of a certain event in two experimental groups. If the number of events is under 5, I also report the Fisher exact p statistic in addition to the p-value of the χ^2 test; the exact p is more reliable in this case. These statistical tests were performed using Statistica 5.0.

We consider a test result significant if p is less or equal 0.1.

3.1.2 Presentation

The presentation of the results uses two forms: The data that underwent formal statistical inference is presented either in tables or directly in the test using probabilities, absolute values, and percentages.

For other data or for additional illustration we use histograms. Since most distributions have only few distinct values, histograms represent the data quite precisely, yet allow for easy consumption and comparison.

3.2 Performance on the Tasks

In this section we compare the performance of the groups with and without PD. For each task, we first consider the individual classes of errors that occurred and then investigate global quantitative effects with respect to time required and solution quality obtained. We also study the learning effect from the first to the second task performed by each subject.

3.2.1 Metrics employed

In the evaluation below, the following measurements and criteria will be used. Each class of them is described by the following terms [5]: A measurement can be either objective (and therefore in principle completely reproducible and out of question) or subjective (and therefore subject to debate); it can be either direct or be derived from other measurements; it can be on a nominal, ordinal, interval, cardinal, or absolute scale; it can have limited precision and limited accuracy even if it is objective.

Groups (objective, direct, nominal scale, completely accurate): The groups (as described in Section 2.2 on page 9) were used for two purposes: Comparing performance with PD against performance without PD and additionally comparing performance in the first task against performance in the second. For instance for the Element task comparing PD against no PD means comparing the union of the groups E^+T^- and T^-E^+ against the union of the groups T^+E^- and $E^-T^+$and comparing each subject's first task against the second means comparing E^+T^- against T^-E^+ (once with respect to *Element* and once with respect to *Tuple*) and E^-T^+ against T^+E^-(likewise).

Incidence counts (subjective/objective, direct, absolute scale): Incidence counts reflect how often a particular event occurs in a group. We considered incidence counts for various classes of errors in the solutions delivered by the subjects. In a few of the cases, it is debatable whether a certain solution is an instance of the event or not, so there is some amount of subjectivity in the data. Except for subjectivity, the incidence data is considered accurate, as we gathered it carefully.

Time measurements (objective, direct, cardinal scale, precision 1 minute, accuracy about 1 minute): The subjects noted start and end times on each page of the experiment materials. We computed the difference between the end of the last page of a task and the start of the first page of the task as the work time measurement; the subjects did not make major breaks that had to be subtracted. We used the time data only on the task level (as opposed to the subtask level) as it is the one with the clearest interpretation and the only one that was validated upon collection of each part of the experiment materials.

Points (subjective/objective, direct, cardinal scale, precision 1 point, completely accurate): We graded the solutions by assigning points, using a penalty system where possible (subtracting a fixed number of points for each kind of error). The individual penalties are explained in the actual results sections below. We consider the differences of numbers of points between the groups for each subtask individually ("points 1, points 2, points 3), for the whole task ("all points"), and for the possibly PD-relevant subtasks ("relevant points", for Element this is points 3, for Tuple it is points 2+points 3). Points are meant to characterize the quality of a solution, but this interpretation must be applied only with care, as the scale chosen is necessarily subjective.

Productivity (subjective/objective, derived, cardinal scale, precision 1 point per hour): A measure that is meant to characterize productivity was derived by computing points per hour. Due to the restrictions of the point measure, points per hour also have to be interpreted with care.

Group filters (objective, derived, nominal scale): For the interpretation of results it is sometimes useful to consider only those subjects that have a certain attribute. In particular we would like to know how the results for talented subjects differ from the less talented ones. For this purpose, we use three different filters for selecting parts of a group:

1. Experience: We consider a subject to have high programming experience, if he has written a largest program of at least 2000 LOC and has used at least 4 programming languages. Otherwise we consider him to have low experience. The above threshold values are the medians of the answer distributions of the respective questions in our first questionnaire. There are 10 subjects with high experience and 12 with low experience.
2. Pattern knowledge: We consider a subject to have good knowledge of design patterns, if he has obtained at least 8 points for answers a.) and b.) in the pattern test (second questionnaire). Otherwise we consider him to have low pattern knowledge. The threshold of 8 is the median of the respective point distribution. Note that the equivalence of many points to a good active knowledge of design patterns might be dubious. It is possible that our test is no good measure of applicable pattern knowledge. There are 13 subjects with high pattern knowledge and 9 with low pattern knowledge.

3.2.2 Task "Element"

As mentioned above, for subtasks 1 and 2 it should not matter whether PD is present or not. For subtask 1, we see the absolute frequency of different kinds of errors in the solutions in Figure 3.1. In the left histogram, the data for the group with PD is shown, on the right without PD. The individual error codes were chosen in an ad-hoc fashion and do not mean anything in particular. In principle, there could be more data points than subjects in the group because each solution can have more than one error. As we see, there is little difference between left and right — just as expected.

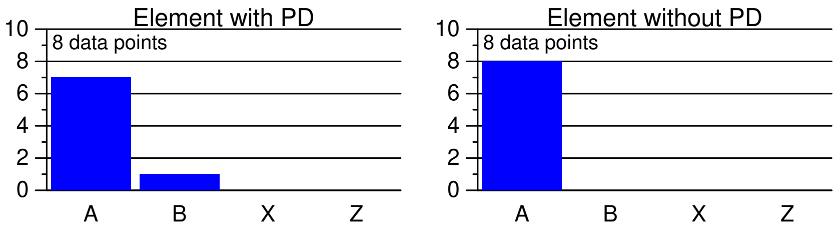


Figure 3.1: Frequency of different errors for subtask 1 of Element.
Codes: A:correct solution B:changes individual strings in main(), X:other, Z:no answer.
A represents no error and costed no points. B costed both points.

The same is true for subtask 2 as shown in Figure 3.2 below. There are much fewer error classes, but still only small differences in their frequency from one group to the other.

For subtask 3, PD was supposed to be relevant. Therefore one might expect to find error class N more frequently in the group without PD; see Figure 3.3 below). However, there is no significant difference.

Next, we aggregate all error classes into a sum of points per subtask by applying the penalties indicated in the figure captions above. We also review the time required and the resulting value of points per hours. For all of these analyses we removed the data points of subjects that did not complete the Element task. Fortunately, there are exactly as many such data points from the E^+ group as from the E^- group, namely 3. These 6 data points are ignored completely in the discussion below.

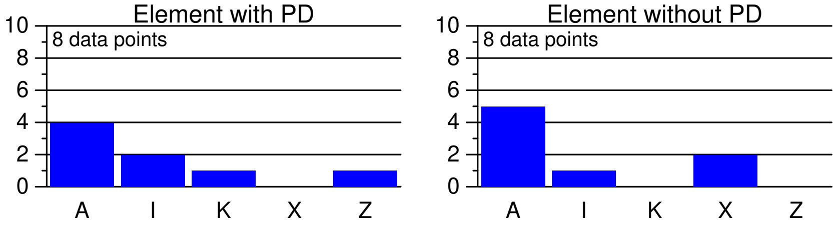


Figure 3.2: Frequency of different errors for subtask 2 of Element.
Codes: A:correct solution, I:counter in print() method in xxElement classes, K:counter elsewhere in xxElement classes, X:other, Z:no answer.
A represents no error and costed no points. I and K costed one point. X was judged individually.

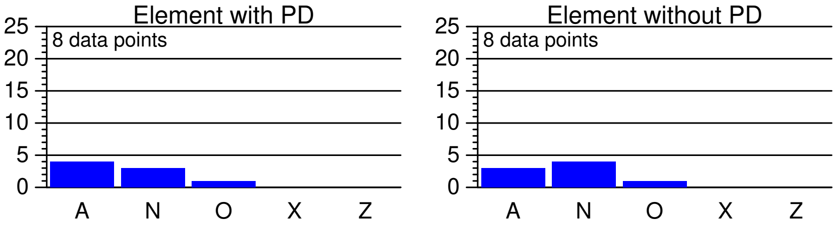


Figure 3.3: Frequency of different errors for subtask 3 (class construction) of Element.
Codes: A:correct solution with new Visitor class N:new method in xxElement classes, O:modifies existing Visitor class instead, X:other, Z:no answer.
A represents no error and costed no points. N costed two points. O costed four points.

The results are presented in Table 3.1 below. Note that the table does not contain some of the lines present in the corresponding table from the Karlsruhe experiment report, due to differences in the experiment design. The results can be summarized as follows:

Points: As expected, there are no differences in the points for subtasks 1 and 2, where PD was not supposed to be relevant (lines 1 and 2). However, there is also no significant difference for the pattern-relevant class construction subtask (line 3). As a result, the differences for all points (line 6) or relevant points (line 7, see also Figures 3.4 and 3.5 on page 26) are not significant as well.

There are similar numbers of correct solutions with and without PD for subtask 3 (4 out of 8 versus 3 out of 8, $\chi^2 = 0.25$, $p = 0.61$, Fisher exact $p = 0.50$).

If we look separately at only those subjects that have high (line 10) or low (line 11) amounts of previous programming experience, or those with good (line 8) or less good (line 9) results in the pattern knowledge test, we find a few differences that appear significant. However, there are so few data points in the respective samples that these results should be considered highly dubious and be discarded, as they are highly susceptible to even modest changes in but a single data point.

Summing up, PD had rather little influence on the number of points. Most of the subjects that did not drop out of the task entirely performed rather well.

Time: The time required for solving Element was significantly lower with PD than without (line 12, see also Figures 3.6 and 3.7 on page 26). This is a rather clear result: The introduction of PD reduces task completion time by about one quarter of its previous value. Again, the results from applying the pattern knowledge and experience filters should be discarded (lines 15 to 18).

	Task Element Variable	best	mean with PD P^+	mean w/o PD P^-	means difference (90% confid.) I	signifi-cance p
1	points 1	2	1.7	2.0	$(-38\% \ldots 0.0\%)$	(0.34)
2	points 2	2	1.3	1.5	$-50\% \ldots 33\%$	0.42
3	points 3	8	6.7	6.5	$-11\% \ldots 19\%$	0.28
6	all points	12	9.8	10.0	$-18\% \ldots 13\%$	0.48
7	relevant points	8	6.7	6.5	$-12\% \ldots 19\%$	0.28
8	— high pat.knwldg.	8	7.1	7.0	$-13\% \ldots 17\%$	0.41
9	— (low pat.knwldg.)	8	4.0	6.0	$-50.0\% \ldots - 16\%$	**0.003**
10	— (high experience)	8	7.1	7.3	$-16\% \ldots 14\%$	0.40
11	— (low experience)	8	4.0	6.0	$-47\% \ldots - 20\%$	**0.000**
12	time (minutes)	29	52.1	67.5	$-43\% \ldots - 0.5\%$	**0.046**
15	— high pat.knwldg.	29	45.5	68.0	$-52\% \ldots - 14\%$	**0.000**
16	— (low pat.knwldg.)	45	98.0	67.0	$31\% \ldots 63\%$	**0.000**
17	— (high experience)	29	45.5	62.0	$-55\% \ldots - 0.3\%$	**0.047**
18	— (low experience)	54	98.0	70.8	$29\% \ldots 49\%$	**0.000**
19	points per hour	20	13.4	9.4	$3.4\% \ldots 81\%$	**0.035**
20	— high pat.knwldg.	20	14.9	9.6	$23\% \ldots 87\%$	**0.001**
21	— (low pat.knwldg.)	14	3.1	9.2	$-101\% \ldots - 38\%$	**0.000**
22	— (high experience)	20	14.9	11.4	$-2.8\% \ldots 66\%$	**0.066**
23	— (low experience)	11	3.1	8.2	$-79\% \ldots - 46\%$	**0.000**

Table 3.1: (left to right:) Name of variable, best result obtained by any subject, 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 (one-sided). "relevant points" are identical to points 3. I and p were computed using resampling with 10000 trials. Parentheses around intervals and p values mean dubious results due to zero variance in at least one of the samples. Parentheses around variable names mean dubious results due to at least one sample having less than four entries (low pat.knwldg. E^+ (E^-) samples have 1 (4) entries, respectively; high experience E^+ (E^-) samples have 7 (3) entries; low experience E^+ (E^-) samples have 1 (5) entries) due to subject mortality.

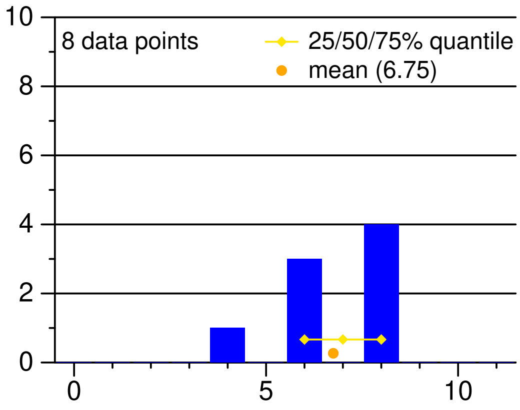


Figure 3.4: Distribution of "relevant points" obtained in task Element with PD.

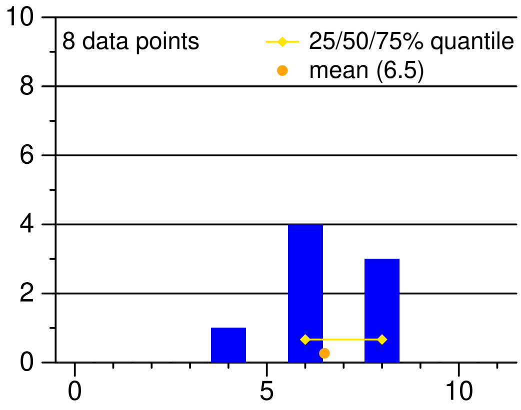


Figure 3.5: Ditto, without PD.

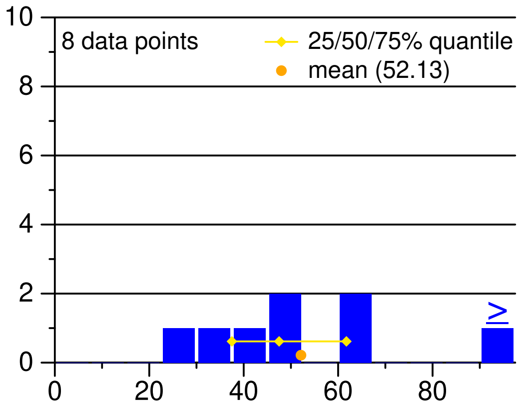


Figure 3.6: Distribution of time (in minutes) required for solving task Element with PD.

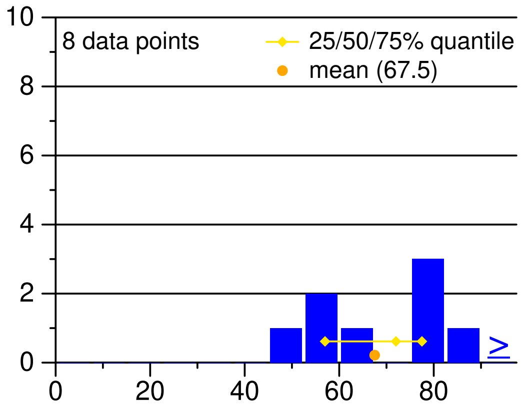


Figure 3.7: Ditto, without PD.

For points per hour, the results are analog to time alone (lines 19 to 23).

3.2.3 Task "Tuple"

In subtask 1 of Tuple, only a single subject made an error, all others were completely correct. For subtask 2 we do also not find any large differences in the number of errors in both groups as shown in Figure 3.8.

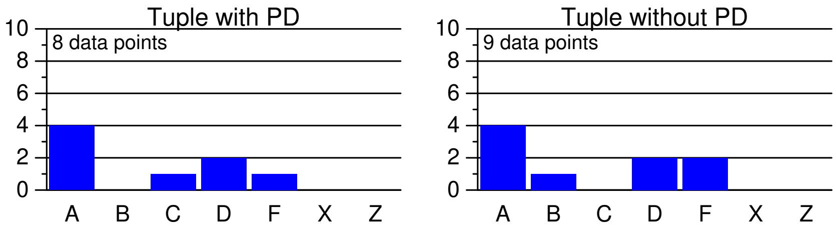


Figure 3.8: Frequency of different errors for subtask 2 (class construction) of Tuple.
Codes: A:correct solution B:inherits from class Name_Number_Tuple_Display_1, C:modifies class Name_Number_Tuple_Display_1, D:output not sorted, F:inherits from class Tuple_Display, X:other, Z:no answer.
A does not represent an error and costed no point. B, D costed two points. C, F costed four points.

Subtask 3 again exhibits more errors in the group with PD, but there is no striking pattern is the distributions (see Figure 3.9).

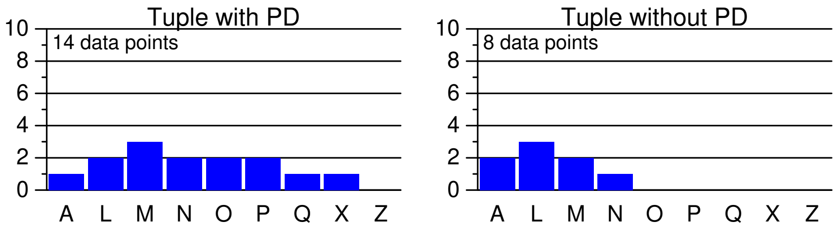


Figure 3.9: Frequency of different errors for subtask 3 (class construction) of Tuple.
Codes: A:correct solution, L:inherits from Tuple_Display, M:inherits from Tuple_Display_A, N:less_than() not implemented, O:format() not implemented, P:select() not implemented, Q:new_tuples() not implemented, X:other, Z:no answer.
A does not represent an error and costed no point. N, O, P, Q, X each costed one point. L, M each costed two points.

Next, we aggregate all error classes into a sum of points per subtask by applying the penalties indicated in the figure captions above. We also review the time required and the resulting value of points per hours. For all of these analyses we removed the data points of subjects that did not complete the Tuple task. Fortunately, there are about as many such data points from the T^+ group as from the T^- group, namely 5 versus 4, so that the mortality does not bias the results a lot. These 9 data points are ignored completely in the discussion below.

The results are presented in Table 3.2 below. Note that the table does not contain some of the lines present in the corresponding table from the Karlsruhe experiment report, due to differences in the experiment design. The results can be summarized as follows:

	Task Tuple Variable	best	mean with PD P^+	 w/o PD P^-	means difference (90% confid.) I	signifi- cance p
1	points 1	2	1.8	2.0	$(-25\% \ldots 0)$	(0.32)
3	points 2	8	5.5	5.8	$-44\% \ldots 29\%$	0.41
5	points 3	8	5.0	6.4	$-44\% \ldots -0.3\%$	**0.048**
6	all points	18	12.3	14.2	$-33\% \ldots 5.6\%$	0.12
7	relevant points	16	10.5	12.2	$-38\% \ldots 8.5\%$	0.15
8	— (high pat.knwldg.)	16	9.3	13.2	$-45\% \ldots -13\%$	**0.001**
9	— (low pat.knwldg.)	16	11.6	10.0	$-20\% \ldots 53\%$	0.26
10	— (high experience)	16	11.0	13.2	$-42\% \ldots 12\%$	0.23
11	— (low experience)	16	10.2	10.0	$-33\% \ldots 40\%$	0.42
12	time (minutes)	29	64.1	62.7	$-23\% \ldots 29\%$	0.45
15	— (high pat.knwldg.)	29	63.6	55.2	$-23\% \ldots 56\%$	0.27
16	— (low pat.knwldg.)	50	64.6	81.5	$-45\% \ldots 3.2\%$	**0.087**
17	— (high experience)	29	55.0	55.2	$-28\% \ldots 25\%$	0.48
18	— (low experience)	50	68.7	81.5	$-40\% \ldots 8.8\%$	0.12
19	points per hour	29	11.8	15.5	$-55\% \ldots 4.0\%$	**0.086**
20	— (high pat.knwldg.)	29	11.6	18.2	$-69\% \ldots -6.3\%$	**0.019**
21	— (low pat.knwldg.)	15	12.1	9.0	$-4.4\% \ldots 67\%$	**0.054**
22	— (high experience)	29	13.5	18.2	$-53\% \ldots -1.7\%$	**0.030**
23	— (low experience)	16	11.0	9.0	$-16\% \ldots 62\%$	0.16

Table 3.2: (left to right:) Name of variable, best result obtained by any subject, 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 (one-sided). "relevant points" are points excluding subtask 1. I and p were computed using resampling with 10000 trials. Parentheses around intervals and p values mean dubious results due to zero variance in at least one of the samples. Parentheses around variable names mean dubious results due to at least one sample having less than four entries (high pat.knwldg. T^+ (T^-) samples have 3 (5) entries, respectively; low pat.knwldg. T^+ (T^-) samples have 3 (2) entries, respectively; high experience T^+ (T^-) samples have 2 (5) entries; low experience T^+ (T^-) samples have 4 (2) entries) due to subject mortality.

Points: As expected, there is no significant difference in the number of points obtained in subtask 1 (line 1). There is also no difference for the class construction subtask 2 implementing an observer using the Template Method superclass (line 2). Surprisingly, there is a significant difference favoring the group *without* PD in the other class construction subtask. This result seems to contradict our hypothesis that PD is helpful. However, we believe the result to be spurious as we will argue below in Chapter 4.

There are similar numbers of correct solutions with and without PD for subtask 2 (3 out of 6 versus 2 out of 7, $\chi^2 = 0.63$, $p = 0.43$, Fisher exact $p = 0.41$) as well as subtask 3 (1 out of 6 versus 2 out of 7, $\chi^2 = 0.26$, $p = 0.61$, Fisher exact $p = 0.56$) and both together (1 out of 6 versus 1 out of 7, $\chi^2 = 0.01$, $p = 0.91$, Fisher exact $p = 0.73$). The small fraction of correct solutions makes it obvious, though, that overall the task was too difficult for these subjects.

Similar statements apply also to the total number of points and the relevant points (lines 6 and 7, see also Figures 3.10 and 3.11).

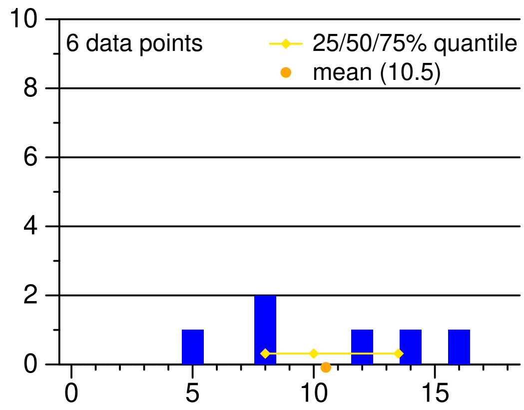


Figure 3.10: Distribution of “relevant points” obtained in task Tuple with PD.

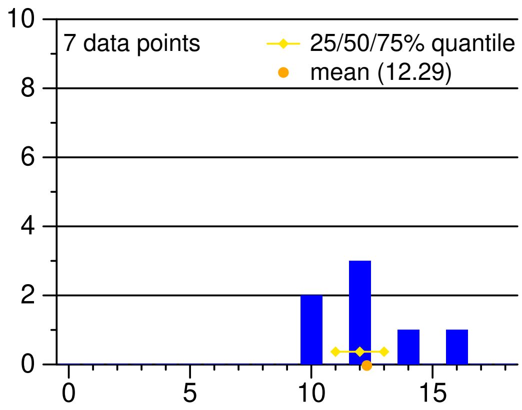


Figure 3.11: Ditto, without PD.

As discussed in the section on the Element task above, applying the pattern knowledge and experience filters for splitting the groups leads to dubious results as the groups become too small (lines 8 to 11). The results should therefore be taken out of account.

Time: For program Tuple, subjects with PD required almost exactly the same amount of time as subjects without PD (line 12, see also Figures 3.12 and 3.13 below). The results for partial groups should again be discarded (lines 15 to 18). Combining points and times into productivity, we find another spurious difference, just like for points alone (line 19).

3.2.4 Learning effect

For the Karlsruhe experiment, we had evaluated the subgroups separately to assess a possible learning effect during the experiment. For instance for the Element task, we compared the group E^+T^- to group E^-T^+ to review performance for the first task performed by each subject only. Likewise for the second task and likewise for Tuple. We also compared the differences of groups with and without PD for the first versus the second task.

In the present experiment, such an analysis is not reasonable: The groups just become too small to be reliable. We therefore leave the analysis out of the present report.

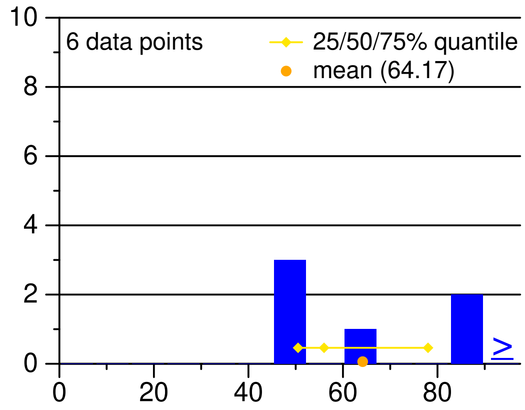


Figure 3.12: Distribution of time (in minutes) required for solving task Tuple with PD.

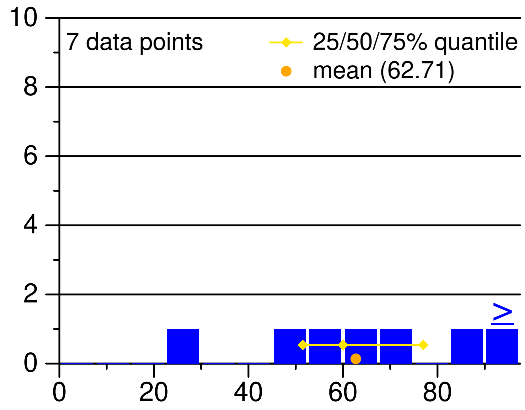


Figure 3.13: Ditto, without PD.

3.3 Underlying Effects

In this section we analyze data from our postmortem questionnaire that might explain some of the effects underlying the results.

3.3.1 Faults in Pattern Recognition

The first question in our postmortem questionnaire concerned the patterns found by the subjects. We see the results for Element in Figures 3.15 and 3.14 below. Not surprisingly, the subjects identified the patterns more reliably when PD was given although the difference is not significant for the Composite pattern (Visitor: 7 out of 8 versus 3 out of 8, $\chi^2 = 4.27$, $p = 0.039$, Fisher exact $p = 0.059$. Composite: 8 out of 8 versus 6 out of 8, $\chi^2 = 2.29$, $p = 0.13$, Fisher exact $p = 0.23$). The number of spurious pattern identifications is smaller with than without PD (3 out of 18 versus 9 out of 19, $\chi^2 = 3.98$, $p = 0.046$, Fisher exact $p = 0.049$). As we see, subjects will not recognize patterns as reliably without PD, even if the program is not overly large or complex.[2].

For Tuple, on the other hand, PD did not make a difference either for correct nor for spurious pattern recognition as shown in Figures 3.17 and 3.16 below; another indication that something went wrong with the Tuple task in the experiment. (Observer: 4 out of 6 versus 5 out of 7, $\chi^2 = 0.03$, $p = 0.85$, Fisher exact $p = 0.66$. Template Method: 5 out of 6 versus 4 out of 7, $\chi^2 = 1.04$, $p = 0.31$, Fisher exact $p = 0.34$. spurious patterns: 4 out of 13 versus 7 out of 16, $\chi^2 = 0.51$, $p = 0.47$, Fisher exact $p = 0.37$).

Check marks decorated with parentheses or question marks were counted as one half. Note that the above numbers exclude the subjects that did not complete the respective task (as opposed to the following figures below, which all include them).

3.3.2 Problem Solving Method

Directly after each task we asked how the subjects had solved them. Unfortunately, the free-text answers to these questions were unusable, as different subjects used different views and different levels of abstraction in

[2]Another interpretation of the results is that some subjects forgot the patterns again before the postmortem questionnaire.

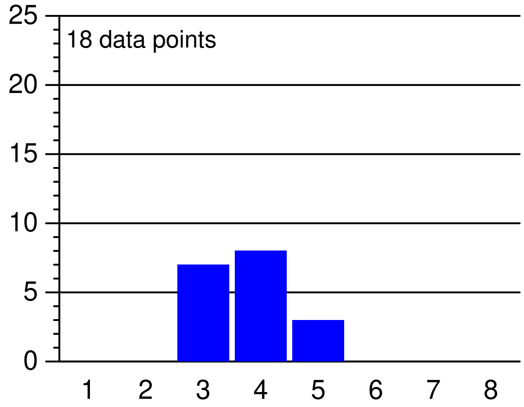


Figure 3.14: Number of times each pattern was checked as “found in Element” by subjects with PD. 1=Command, 2=Observer, 3=Visitor, 4=Composite, 5=Template Method, 6=Strategy, 7=Mediator, 8=Chain of Responsibility.

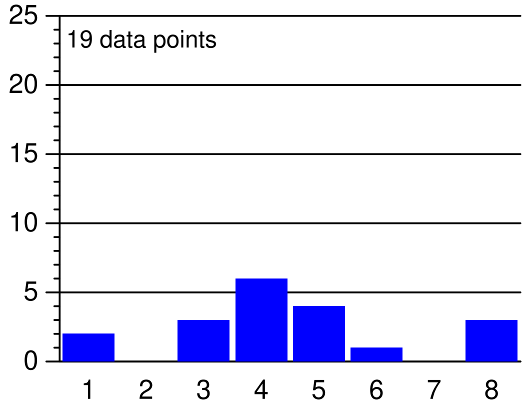


Figure 3.15: Ditto for subjects without PD. In all cases, checkmarks that were accompanied by question marks or question marks alone were counted as 0.5.

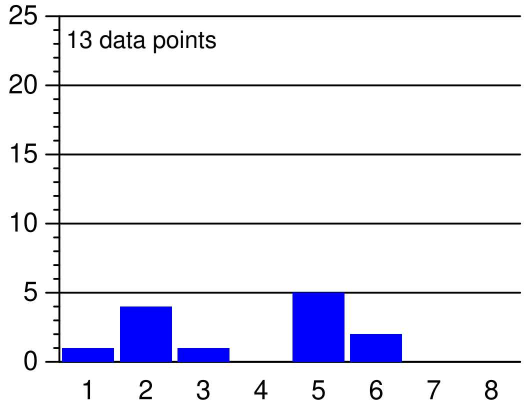


Figure 3.16: Number of times each pattern was checked as “found in Tuple” by subjects with PD. Encoding as in Figure 3.14.

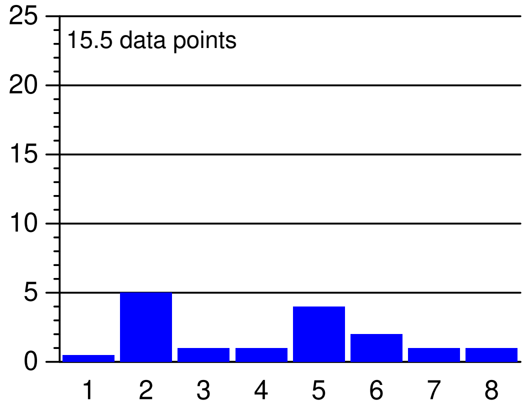


Figure 3.17: Ditto for subjects without PD.

their answers. However, there also was a multiple-choice question in our postmortem questionnaire concerning whether, when, and why subjects had actively searched for design patterns in the programs.

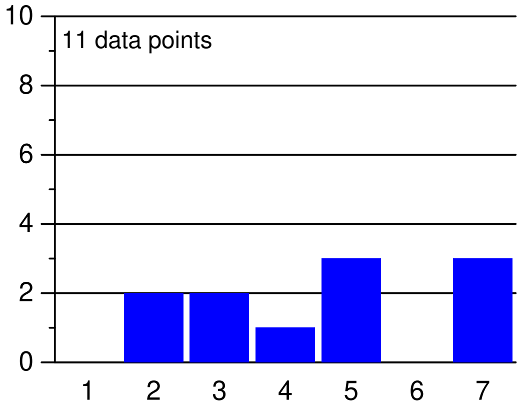


Figure 3.18: Element with PD: Whether, why, and when subjects actively searched for patterns. 1=No, unnecessary to know patterns; 2=No, patterns were documented; 3=No, found them immediately; 4=No, other reason; 5=Yes, right from the start; 6=Yes, when it became my only chance; 7=Yes, later for other reason.

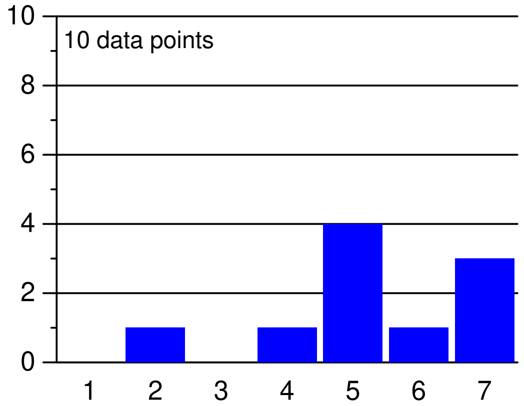


Figure 3.19: Ditto for subjects without PD.

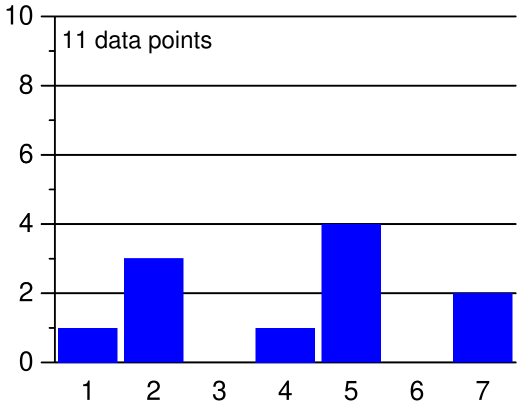


Figure 3.20: Tuple with PD: Whether, why, and when subjects actively searched for patterns. Encoding as in Figure 3.19.

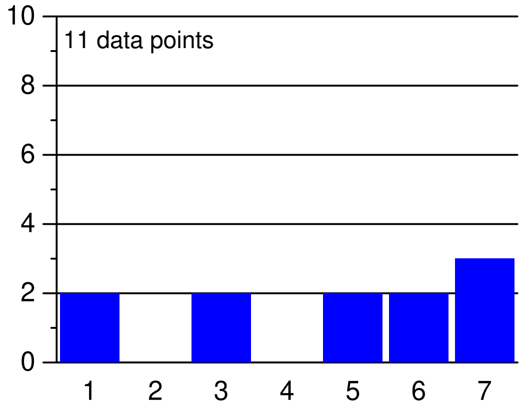


Figure 3.21: Ditto for subjects without PD.

The results are shown in Figures 3.18 to 3.21. For Element/Tuple, as many as 55%/55% of the subjects with PD stated that they actively searched for patterns, whereas without PD these numbers are 20%/36%. Moreover, it is striking that so few of the subjects with PD chose the expected answer 2 “I did not search for patterns, because the patterns were documented.” These two observations may mean that for some reason the question was difficult to interpret for the subjects. As for a learning effect, see Figures 3.22 and 3.23 below. Apparently only few subjects consciously learned that searching for the patterns might be helpful.

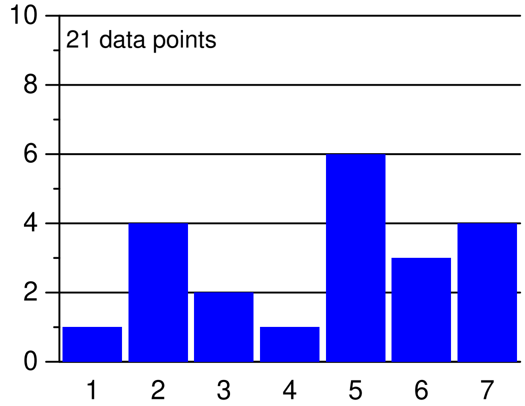


Figure 3.22: Whether, why, and when subjects actively searched for patterns in their first task. Encoding as in Figure 3.19.

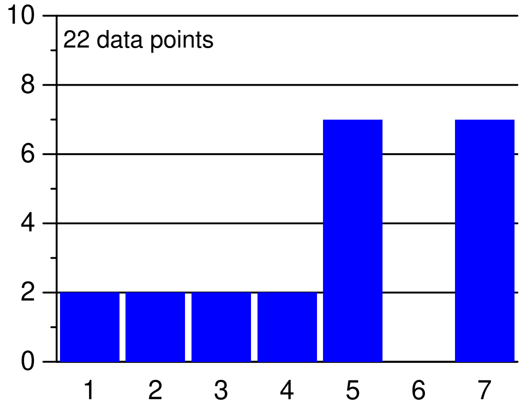


Figure 3.23: Ditto for second task.

3.4 Subjects' Experiences

3.4.1 Difficulty of tasks

Our question on how difficult the subjects found the tasks had the following results, see Figure 3.24 below:[3] Little more than one quarter of the subjects found the tasks difficult and on average the subjects found the second task somewhat easier than the first — two opinions that are not backed up by the objective results. For the overall difficulty the histograms show that some "difficult" answers are reduced to "somewhat difficult" when PD is introduced, but overall it did (subjectively!) make only a modest difference whether PD was present or not.

How well the subjects were subjectively able to concentrate on the tasks is shown in Figure 3.25 below. Generally, our subjects could, so they claimed, concentrate sufficiently well; not much worse in the second task than in the first. Interestingly, the presence or absence of PD had no impact.

We also asked how many errors the subjects thought they had in their solutions. The results are shown in Figure 3.26 on page 35. Generally, the subjects had no high confidence in their solutions. Two thirds or more of them expected to have at least one error. Confidence did not change from the first task to the second, but was a little *lower* with PD than without.

These confidence ratings are far more negative than the difficulty ratings would suggest. This discrepancy may mean that at least the difficulty ratings are dubious.

3.4.2 Is Pattern Knowledge Helpful?

Two further questions concerned a subjective estimation whether knowledge of design patterns was useful for solving the tasks.

The first question asked for the usefulness of design pattern knowledge in general. The results are shown in Figure 3.27 on page 35. From all aspects, a majority of the subjects found previous pattern knowledge useful:

[3] Again, we treat the ordinal scale of the answers as an interval scale.

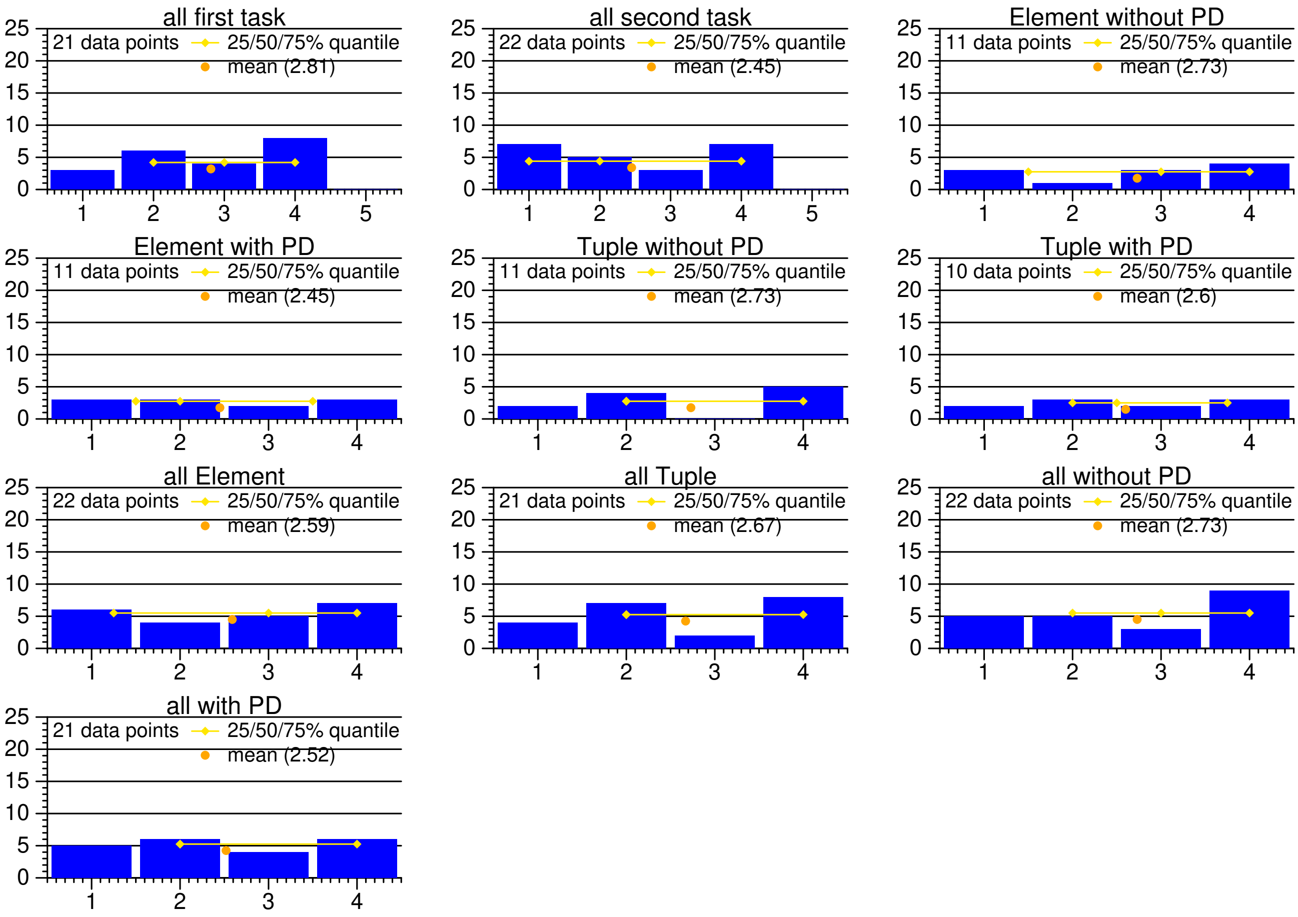


Figure 3.24: Subjective difficulty of tasks. 1=quite simple, 2=not too simple, 3=somewhat difficult, 4=difficult.

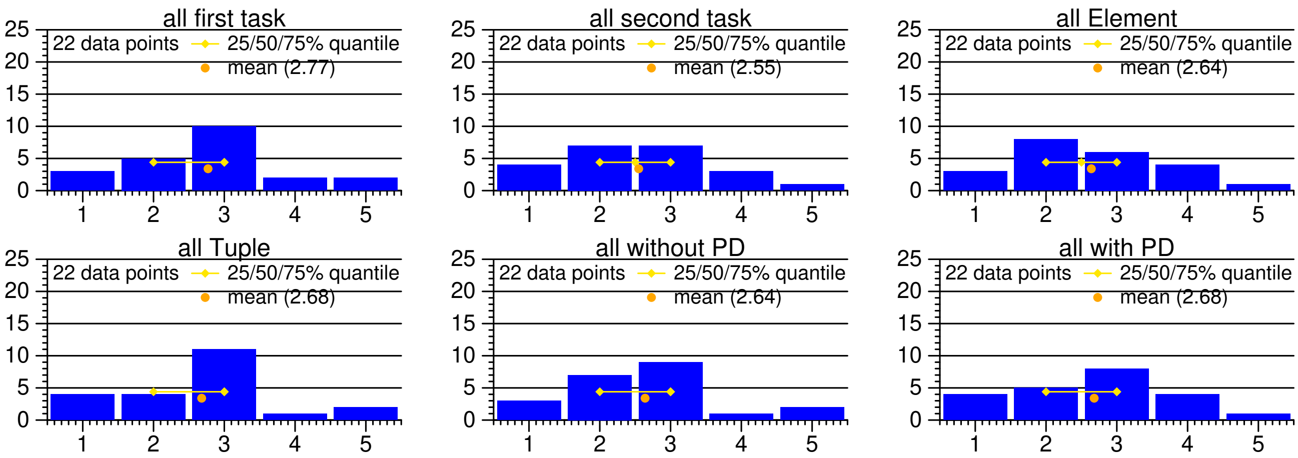


Figure 3.25: Subjective concentration ability during tasks. 1=very high, 2=high, 3=OK, 4=somewhat low, 5=low.

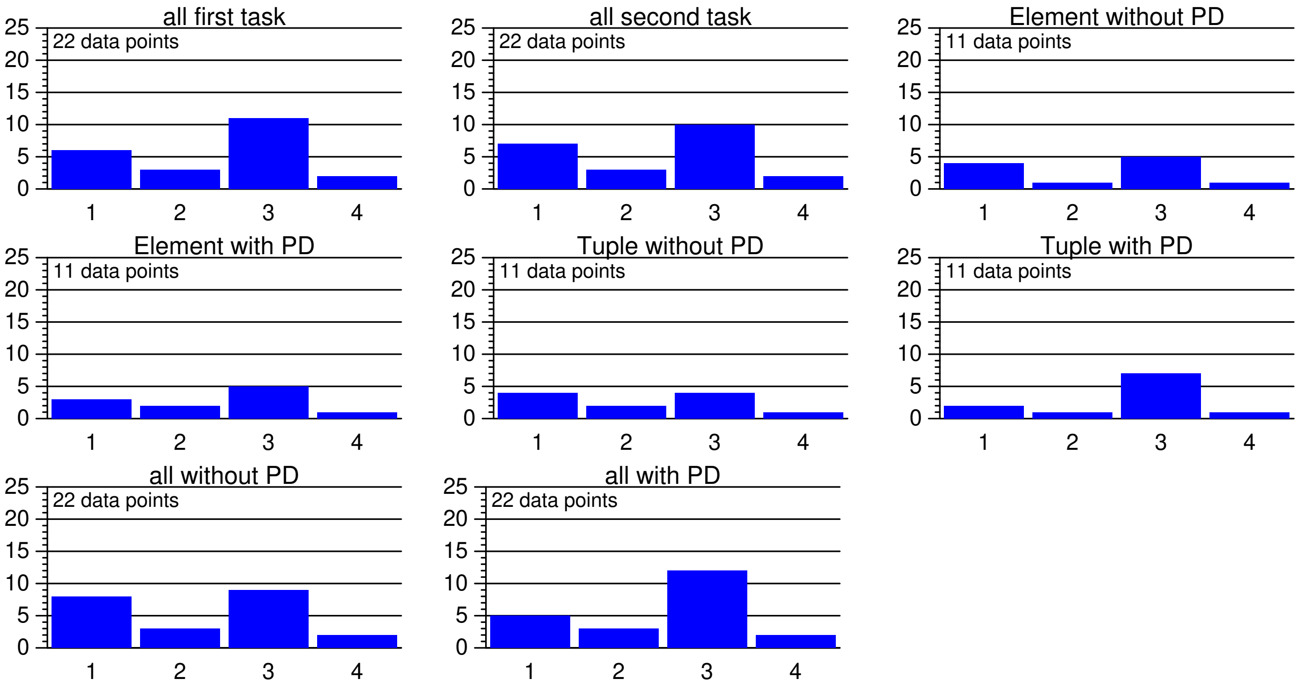


Figure 3.26: Subjects' estimate of errors in solutions. 1=none, 2=at most one,3=several, 4=don't know.

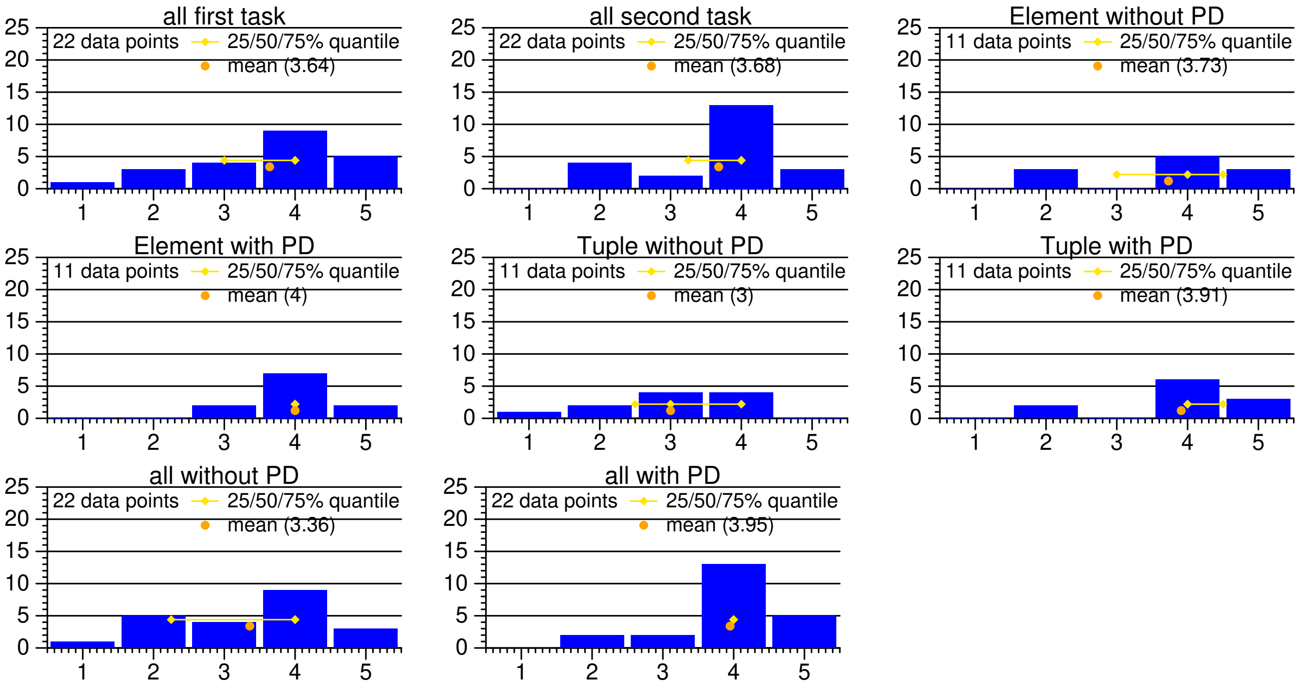


Figure 3.27: Subjective helpfulness of previous pattern knowledge for tasks. 1=no, 2=little, 3=am unsure, 4=yes, 5=yes much.

This is true for the first task as well as the second, for Element as well as for Tuple, and with PD given as well as without. However, without PD the usefulness was considered lower than with PD, in particular for Element. The latter is another positive result: as the programs were the same, this result (if correct) indicates that PD allows for better exploitation of pattern knowledge.

3.4.3 Is Pattern Documentation (PD) Helpful?

The second question asked for the usefulness of the concrete PD given in the programs. The results are shown in Figure 3.28. In principle, an answer to this question makes sense only for the cases with PD. Most of the

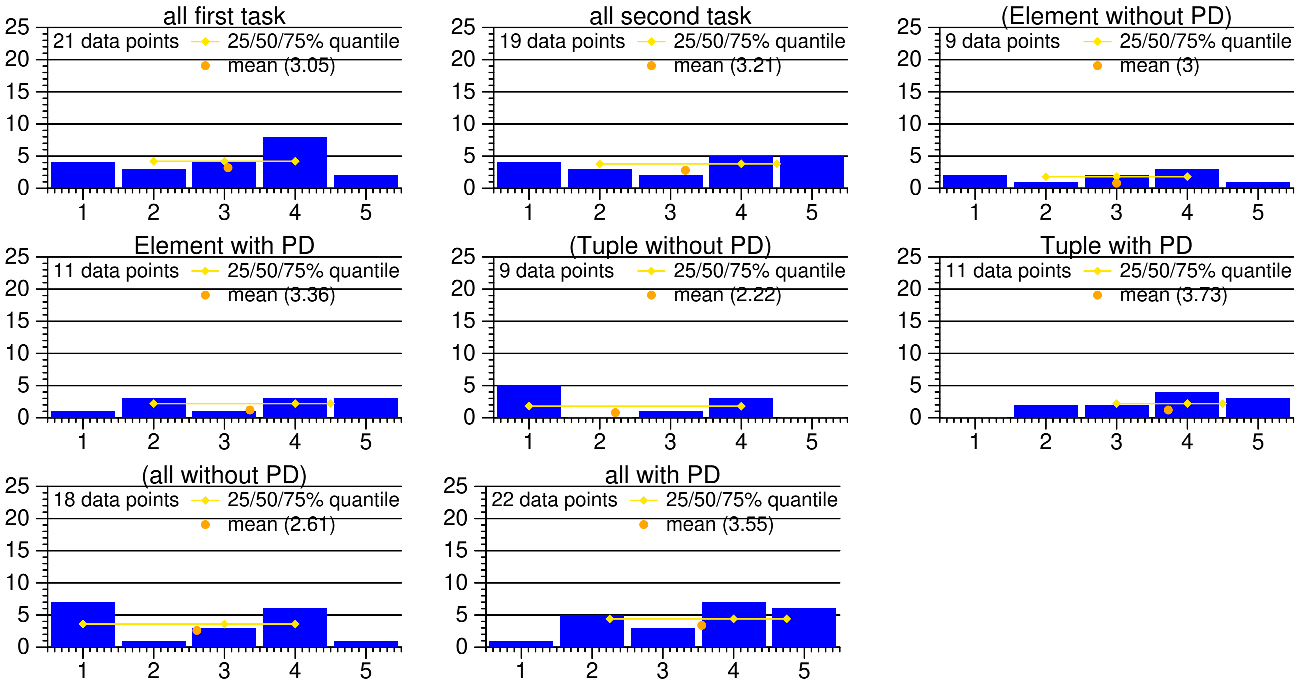


Figure 3.28: Subjective helpfulness of pattern documentation in the programs. 1=no, 2=little, 3=am unsure, 4=yes, 5=yes much.

subjects answered even without PD, though, probably meaning "would have been useful". A majority of the subjects with PD found it helpful, but there is also a significant number of subjects who think otherwise. Only a small number found it very helpful. There are significantly more "not useful" answers in the group without PD than in the group with PD (7 out of 18 versus 1 out of 22, $\chi^2 = 7.30$, $p = 0.0069$, Fisher exact $p = 0.0097$).

Chapter 4

Conclusion

The design of this experiment was very conservative; many design decisions biased the experiment towards *not* showing any effects from adding PD (see also the discussion of external validity in Section 2.8 on page 19):

1. The subjects knew they would participate in an experiment "about design patterns", so they were well motivated to find patterns in the programs. In many cases, this may have made PD superfluous and reduced its apparent benefits.
2. The programs were rather small, so even without PD the subjects could achieve program understanding within a reasonable time. Again, in reality PD will be more helpful for pattern-relevant tasks as the program understanding effort that it can save grows with the size of the program.
3. 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. In industrial reality a smaller fraction of tasks will be pattern-relevant, but for those that are, patterns would be correspondingly harder to exploit without PD, as the patterns are less frequent (thus more surprising) and are buried in a host of other details.
4. 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. In contrast, most programs in the real world lack sufficient design information. PD might be a good means to improve design documentation, as it is rather compact and easy to provide.

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.

In fact we find that our results for the Element task support our hypothesis H1 ("PD makes changes faster"), but is inconclusive with respect to H2 ("PD reduces mistakes"), see Section 2.1 on page 8):

For the Element task, the quality of the solutions was about the same with and without PD given, but the group with PD was about 25% faster. This result neatly complements the Karlsruhe experiment, which supported H2 but was inconclusive for H1 in the Element task.

The Tuple task had a rather clear result (supporting H2, inconclusive for H1) in the Karlsruhe experiment. On first look, the Tuple data of the present experiment appears to be inconclusive with respect to H1 and appears to indicate that the opposite of H1 might be true: that PD actually hampers maintenance. We believe that these results are due to unfortunate circumstances and have to be discarded altogether for the following reasons:

1. The hampering effect mentioned above is completely implausible. There is no explanation why such a small amount of additional information as the PD present in our Tuple program (2.5% of the lines) should have such a negative effect. The only plausible explanation is as an artifact of the experimental situation: some subjects vainly tried to understand the documented pattern for a long time instead of ignoring the PD. This would probably not happen in reality.

2. The rather small fraction of completely correct solutions indicates that the task was in fact too difficult.

3. The large fraction of drop-outs (9 out of 22 subject, or 41%) indicates the same.

We believe that for the Tuple task the pattern knowledge of our subjects was simply too low. The negative influence that PD had on solution quality is probably due to an effect of the sort "Oh damn, there is an Observer pattern and I don't really understand it!"[1] This effect happened to the group with PD but not to the group without PD. Our hypothesis is corroborated by the observation that the solution quality difference between the group almost disappears for subtask 2 (in which the Template Method was relevant); the Template Method pattern was far better understood than the Observer pattern according to our pattern knowledge questionnaire (see Figure 2.13 on page 15).

Probably one main problem with the Tuple task in this experiment was the form of the Observers: No standard GUI representation was possible due to the subjects' missing GUI knowledge. Therefore, all our Observer objects just produced text output lines on a single common output stream, which is quite different from the functionality and appearance with which Observers are usually presented when taught. Thus, the Tuple program was discouraging for the group with PD who knew they should recognize and understand an Observer in it but could not quite do so.

However, concluding from both the Karlsruhe and the new experiment together, we recommend that usage of design patterns routinely be documented in program source code, as the benefits appear to be rather large compared to the effort invested.

Further work should perform similar experiments in different settings, in particular using more difficult tasks, to see whether the benefits from PD are really more pronounced then.

Acknowledgements

Barbara Unger conducted the experiment and Doug Schmidt made it possible by providing the subjects. Michael Philippsen helped during experiment design. Doug Schmidt converted the programs from Java to C++. Barbara Unger and Lutz Prechelt performed the evaluation of the experiment and Michael Philippsen and Walter Tichy (the latter of whom also suggested the whole thing) helped with the interpretation of the results. Lutz Prechelt wrote most of the report.

[1] If this effect is really the reason, the observation is a good motivation for *teaching* design patterns more thoroughly so that they will be understood more readily when they appear in programs.

Appendix A

Tasks and solutions

A.1 Handling Description, Solutions

This appendix contains the original questionnaire administered to the subjects. The questionnaire was handed out in five parts:

1. a part about personal information and subjective knowledge of design patterns
2. design patterns test questions
3. the explanation of the first task (handed out together with the corresponding program file)
4. the explanation of the second task (handed out together with the corresponding program file)
5. a posttest questionnaire

The subjects had to give each part back to the experimenters when they received the next part. Each subject could promptly do this at any time.

The page breaks are similar to those used in the experiment; a few empty pages that were present to give subjects enough room for their answers are left out here.

Here is a very compact description of the correct solutions for the design pattern test questions and for the first and second task:

- Pattern question a:
  See Table 2.2 on page 16
- Pattern question b:
  "Introducing a new method in all classes to be visited"
- Pattern question c:
  "Both allow late specification of some part of a method, but with Template Method the missing part is fixed at object creation time, whereas for Strategy it can still be changed later on."

- Pattern question d:
  "Introduce a separate abstract `Container` superclass that introduces these operations and leave them out of the Composite top class."

- Tuple subtask 1:
  Change line 241 (in `Name_Number_Tuple_Display_1::newTuple()`)

- Tuple subtask 2:
  First, introduce a new class `Name_Number_Tuple_Display_N` by subclassing `Name_Number_Tuple_Display_A`; in particular implement an appropriate `select()` method. Second, call `Tuple_Display *disp2 = new Name_Number_Tuple_Display_N (String ("nonlocal")); tuple_set.new_display (disp2);`

- Tuple subtask 3:
  Introduce a new class `Name_Number_Tuple_Display_I` by subclassing `Tuple_Display`. The new class is exactly like `Name_Number_Tuple_Display_1` except for line 245, which must use `enqueue_head()` instead of `enqueue_tail()`. Subclassing `Name_Number_Tuple_Display_A` does not work, because the sorting mechanism does not allow for maintaining the (reverse) original order of the entries.

- in Element subtask 1:
  Change line 305 (in `AndElement::variants()`)

- in Element subtask 2:
  `u.variants().size()`

- in Element subtask 3:
  Introduce a new class `Variants` as a subclass of `Element_Action`. It has a single `int` instance variable which is set to 1 at each leaf. The subtree values are added in `OR` nodes and multiplied in `AND` nodes. Note that one has to think about what the method bodies would look like or may be tempted to include an `iterate()` auxiliary method as in the pre-existing `Depth` visitor, although such a method is not useful here.

A.2 Original Questionnaire

Instructions and Questionnaire for the C++ - Experiment

Douglas C. Schmidt, Barbara Unger, Lutz Prechelt
Washington University

May 6, 1997

Student Id:

(Please do not write here)

Task	Points
E	
S	
Ms	
Ma	
Mb	
Mc	
Md	
T1	
T2	
T3	
E1	
E2	
E3	

Questionnaire Part 1: Personal Information

Please fill in this part of the questionnaire before reading further and before the actual experiment starts. Enter data or check the boxes as appropriate. *All information will be considered confidential.*

Completeness and correctness of your information (please print!) are important for the accuracy of the scientific results of the experiment. Therefore, please answer all questions.

______________________ ______________________
lastname firstname

______________ __________
student id sex: m/f

______________ __________
today's date time

☐ I am ____________ major in my _____-th term,
subject
☐ undergraduate, ☐ graduate.

Before the CS242 course started I had the following programming experience (altogether):

☐ only theoretical knowledge

☐ wrote less than 300 lines of code myself

☐ wrote less than 3,000 lines of code myself

☐ wrote less than 30,000 lines of code myself

☐ wrote more than 30,000 lines of code myself

I have been programming for about _____ years now and used predominantly the following programming languages (ordered by decreasing experience):

____
language 1, language 2, ...

Before the CS242 course started I had the following experience in object oriented programming:

- ☐ no knowledge
- ☐ only theoretical knowledge
- ☐ wrote less than 300 lines of code myself
- ☐ wrote less than 3,000 lines of code myself
- ☐ wrote less than 30,000 lines of code myself
- ☐ wrote more than 30,000 lines of code myself

Before the CS242 course started I had the following experience in programming graphical user interfaces:

- ☐ no knowledge
- ☐ only theoretical knowledge
- ☐ wrote less than 300 lines of code myself
- ☐ wrote less than 3,000 lines of code myself
- ☐ wrote less than 30,000 lines of code myself
- ☐ wrote more than 30,000 lines of code myself

The longest program that I have written *alone* had about ____________ (LOC) lines of code. It was written in ____________________ (programming language) and consumed an effort of ______________ (person months) person months.

Answer the following question only if you have already worked in a team software project or are doing this currently. The largest program in whose construction I have *participated* had about ______________ (LOC) lines of code altogether and consumed an effort of ______________ (person months) person months. My own contribution was about ______________ (LOC) lines of code or ______________ (person months) person months, respectively.

My understanding of design patterns is as follows:
(Enter a number between 1 and 5 for each design pattern. The number indicates how well you subjectively believe you understand the design pattern.
1: I understand and apply it very well,
2: I understand it well,
3: I understand it roughly,
4: I am beginning to understand it,
5: I do not understand it)

______ Abstract Factory.

______ Command.

______ Observer.

______ Visitor.

______ Bridge.

______ Factory Method.

______ Iterator.

______ Composite.

______ Proxy.

______ Template Method.

______ Strategy.

______ Mediator.

______ Chain of Responsibility.

Now please enter Student Id and time and request new materials from the experimentors.

Student Id: Time:

Please enter the time here when you first read this page thoroughly. Time:

Questionnaire Part 2: Design Patterns

a.) Each column of the table below contains one design pattern and each row contains the name of an operation that a method in that design pattern might perform. The actual name of the method in an instance of the design pattern will usually be different; the given name only indicates the purpose of the method.

On each line, mark all those patterns that usually have the respective operation. (The full name of the abbreviated pattern is “Chain of Responsibility”.)

Operation	Com mand	Obser ver	Visi tor	Com posite	Tem plate Meth.	Stra tegy	Medi ator	Chain of Re- spons.
accept()								
register()								
execute()								
add()								
notify()								
update()								

Shortly answer the following questions.

b.) What is the alternative of introducing a Visitor?

c.) Characterize the differences between “Template Method” and “Strategy”.

d.) In a “Composite”, how can one avoid the problem that even the leaf classes have those operations that are useful only for containers?

Now please enter Student Id and time and request new materials from the experimentors.

Student Id: Time:

First enhance your PATH-Variable with the command:

`source /project/adaptive/cs242/scripts/setpath2`

In these directories are the scripts you need.

Please invoke the script:

`doit`

The script prepares your account. The files you need will be copied in the right directories and a new tmp directory will be created.

If there are any problems or error messages please check with Dr. Schmidt.

What are the scripts doing:
`setpath1, setpath2` enhances your pathes (`PATH, LD_LIBRARY_PATH`) and adds the `package sc_3.1`
`doit` copys the files you need for the environment in your directories
`emto1, emto2, eotm1, eotm2` copys the source code to the specified directories.
`check_in` writes the specified file with a different name (the exercise number is appended)
`undoit` removes all files that the script copied in your directory

Please enter the time here when you first read this page thoroughly. Time:

Task "Tuple"

First you have to type
`eotm2`
and change the directory to
`cd ~/exp2`
Now you have a program in this directory called ***Tuple.C***. Start an `emacs` with this program.

Functionality: Whenever a tuple consisting of a lastname firstname and a phone number is typed in on stdin, the program reads the data. All these records are displayed together on stdout in different styles. Only one output style is implemented yet, called ("chronological"). It shows the records in the order in which they were entered.

Understand this program well enough to perform the tasks described below. If you think that no understanding of certain program parts is required, you need not look at these parts.

Please enter the time here. Time:

Assignments for "Tuple"

1. Change the program so that the commas between lastname and phone number become colons in the "chronological" presentation.

When you have finished this task type
`check_in 1a Tuple.C`

2. Extend the program. Generate a second output style "nonlocal" in which (sorted by lastname) only those Tuples are displayed whose phone number does not start with a 314.

When you have finished this task type
`check_in 2a Tuple.C`

3. Extend the program. Write another TupleDisplay class Name_Number_TupleDisplay_invers that displays Name_Number_Tuple in the format
`lastname, firstname; phone number`
in inversely chronological order (that is, the youngest Tuple at the top; you can use the method
`void enqueue_head(Tuple *t)`
from the class Tuple_Queue).

When you have finished this task type
`check_in 3a Tuple.C`

Now please enter Student Id and time and request new materials from the experimentors.

Student Id: Time:

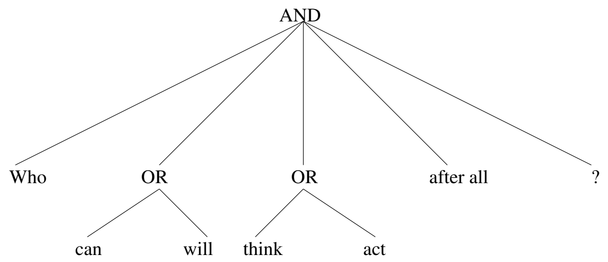
Please enter the time here when you first read this page thoroughly. Time:

Task “Element”

First you have to type
eotm1
and change the directory to
~/exp1
Now you have a program in this directory called **Element.C**. Start an emacs with this program.

Functionality: The program contains a library for constructing And/Or trees. Each leaf of such a tree contains a String. If we interpret And as concatenation and Or as alternation such a tree defines a set of Strings. This set of Strings is called the *variants* of the tree and is computed by the method variants(). Other methods compute a compactly encoded representation of the tree or compute the maximum depth of And nodes, Or nodes, and leafs. The main program generates such a tree and prints its 4 variants, then its compact representation.

The tree generated by the program has the following structure:



The program generates the following output:

```
Who can think after all?
Who can act after all?
Who will think after all?
Who will act after all?
AND ( Who & OR ( can | will ) & OR ( think | act) & after all & ?)
```

Understand this program well enough to perform the tasks described below. If you think that no understanding of certain program parts is required, you need not look at these parts.

Please enter the time here. Time:

Assignments for “Element”

1. Insert an additional space character between those parts of each variant that were combined by concatenation.

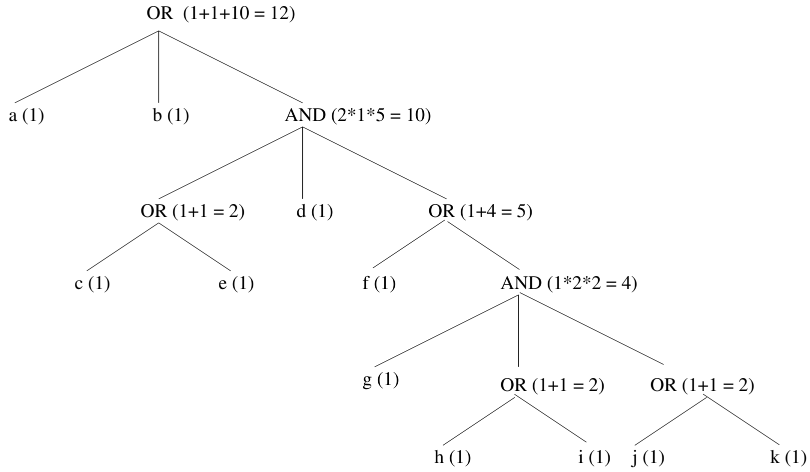
When you have finished this task type
check_in 1a Element.C

2. Insert a statement that computes the *number of variants* of u after u was generated in main().

When you have finished this task type
`check_in 2a Element.C`

3. Extend the program. The above way of computing the number of variants is inefficient, because it may generate a rather large data object that is not required.
Write a program extension that computes the number of variants *without* generating such a large data object.

The algorithm of computing the number of variants is: If you are in a leaf the number of variants is 1, in an Or node it is the sum of all variants of the branches and in an And node it is the product of all variants of the branches. Here is an example:



When you have finished this task type
`check_in 3a Element.C`

Now please enter Student Id and time and request new materials from the experimentors.

Student Id: Time:

Please enter the time here when you first read this page thoroughly. Time:

Questionnaire Part 3: Your Experience in the experiment

This part of the questionnaire is meant to complement the answers you gave above by subjective background information in order to allow for a better analysis of the experiment results.

Most questions have some additional space below for arbitrary comments; your comments will be very useful for us.

Check which of the following design patterns you think occured in one of the programs. If you are unsure, insert a question mark instead of a check mark.

Task	Com mand	Obser ver	Visi tor	Compo site	Tem plate Meth.	Stra tegy	Medi ator	Chain of Resp.
Task "Element"								
Task "Tuple"								

Comment:

The questions below have two checkboxes for each answer:
E for task "Element" and
T for task "Tuple".
So please check exactly one box per column for each question.

Overall and in the given situation I found task "Element"/"Tuple"

- E T pretty simple.
- E T not quite so simple.
- E T pretty difficult
- E T difficult.

Comment:

During task "Element"/"Tuple" my concentration ability was

- E T very high.
- E T high.
- E T OK.
- E T not so high.
- E T low.

Comment:

I believe that my solutions for task “Element”/“Tuple” have

- E T no more errors or omissions.
- E T at most one error or omission.
- E T probably several errors or omissions.
- E T (don’t know).

Comment:

I think that for solving “Element”/“Tuple” my previous knowledge of design patterns was helpful.

- E T No, not at all.
- E T Only a little.
- E T Can’t decide.
- E T Yes, somewhat.
- E T Yes, very much so.

Comment:

I think that for solving “Element”/“Tuple” the labeling, if any, of design patterns in the programs was helpful.

- E T No, not at all.
- E T Only a little.
- E T Can’t decide.
- E T Yes, somewhat.
- E T Yes, very much so.

Comment:

For solving “Element”/“Tuple” I have actively searched for design patterns in the programs.

- E T No, because I did not find it necessary to find or recognize the design patterns used.
- E T No, because obviously the design patterns were documented.
- E T No, because I found the design patterns immediately.
- E T No, because

  ____
  reason for “Element”

  ____
  reason for “Tuple”
- E T Yes, right from the start.
- E T Yes, but only after nothing else seemed to help.
- E T Yes, but only after a while, because

  ____
  reason for “Element”

  ____
  reason for “Tuple”

Comment:

Something else I would like to say (e.g. what I found particularly difficult, unclever in the experimental setup, interesting etc.):

Please invoke now the script `undoit`

Thank you!

Many thanks for participating in our experiment. We hope you learned as much as we did.

Again please enter Student Id and time and return all materials to the experimentors.

Student Id: Time:

Appendix B

Experiment program listings

This appendix contains listings of the the programs given to the subjects during the experiment. The programs use comment conventions introduced in the course. Each program was provided as an ASCII file when the subjects executed the copying script as indicated in the task description.

The listings are given here in the version with PD. The corresponding versions without PD are exactly the same except that in any comment that has a PD marker the part of the comment from the PD marker to the end of the comment would be missing. The PD marker is **** DESIGN PATTERN: ****

B.1 Program "Tuple"

```
/*

   ##################################
   This program is enhanced by:
        PUT IN YOUR NAME!!!
   ##################################

   This program manages sets of tuples.

   A tuple consists of several fields, e.g., last name, first name,
   telephone number.  There are several such tuple types and for each of
   them there is a tuple set type.

   Furthermore, the program contains classes for displaying tuple sets of
   a particular tuple type.  Currently only the tuple type
   Name_Number_Tuple is fully supported.

   This is the class hierarchy:

     abstract class Tuple
         class Name_Number_Tuple
```

```

  class Tupleset

  abstract class Tuple_Display
      class Name_Number_Tuple_Display_1
      abstract class Tuple_Display_A
          class Name_Number_Tuple_Display_2

Note, do not worry about memory leaks in this code.

 *** DESIGN PATTERN: ***

 <Tuple_Display> is the abstract Superclass of **Observer**
 for the data structure <Tupleset>.

 <Tuple_Display_A> is the abstract Superclass of a number of such
 observers that differ with respect to selection, ordering, and
 formatting of the tuples to be shown.  A **Template Method**
 is used to vary these aspects.  */

// Forward declarations.
class Tuple;
class Tuple_Display;
class Tupleset;

// These are necessary to provide String and Queue abstractions.

#include "ace/SString.h"
#include "ace/Containers.h"

// Some typedefs.
typedef ACE_CString String;
typedef ACE_Unbounded_Queue<Tuple *> Tuple_Queue;
typedef ACE_Unbounded_Queue_Iterator<Tuple *> Tuple_Queue_Iterator;
typedef ACE_Unbounded_Queue<Tuple_Display *> Tuple_Display_Table;
typedef ACE_Unbounded_Queue_Iterator<Tuple_Display *> Tuple_Display_Table_Iterator;
typedef ACE_Unbounded_Queue<String> Tuple_String_Vector;
typedef ACE_Unbounded_Queue_Iterator<String> Tuple_String_Vector_Iterator;

class Tuple_Display
  // = TITLE
  //
  // Tuple_Display shows multiple Tuples on the output stream.  Each
  // subclass defines its own presentation style.  Tuple_Displays
  // guarantee that they do not modify the Tuple objects (so that it
  // is possible to commit originals to the Tuple_Display's custody
```

```
  // instead of creating copies).  A program may create a number of
  // Tuple_Display objects and give the same Tuple objects to each of
  // them.
{
public:
  Tuple_Display (const String &name)
    : display_name_ (name)
    {
    }

  // Is called by the user of the <Tuple_Display> to announce that a
  // new <Tuple> should be added to the ones already displayed.
  virtual void new_tuple (Tuple *t) = 0;

  // Is called by the user of the <Tuple_Display> to announce that
  // multiple new <Tuples> shall be added to the ones already
  // displayed.  The caller guarantees that all objects in the
  // <Tuple_Queue> are <Tuple> objects.  <new_tuples> does not change
  // <tq>.
  virtual void new_tuples (Tuple_Queue &tq) = 0;

protected:
  // Queue of tuples.
  Tuple_Queue tuple_queue_;

  // Name of the display;
  String display_name_;
};

class Tupleset
  // = TITLE
  //
  //     Manages a set of <Tuples> and allows individual Tuples to be
  //     added to this <Set>.  Also manages a set of <Tuple_Displays>,
  //     which are notified when a <Tuple> is added.
{
public:
  // Called by a <Tuple> generator to bring a new <Tuple> into the
  // <Tupleset>.
  void new_tuple (Tuple *t)
    {
      // Store <t> at the end of the queue.
      tuple_queue_.enqueue_tail (t);

      Tuple_Display **td = 0;

```

```
        // Inform each <Tuple_Display> that a new tuple has arrived.
        for (Tuple_Display_Table_Iterator iter (this→tuple_display_table_);
                iter.next (td) ≠ 0;
                iter.advance ())
            (*td)→new_tuple (t);
      }

  // Add a new display.
  void new_display (Tuple_Display *td)
    {
      // Store <td> at the end of the queue.
      this→tuple_display_table_.enqueue_tail (td);

      // Inform the new display about all the existing <Tuples>.
      td→new_tuples (this→tuple_queue_);
    }

private:
  // Registered Displays.
  Tuple_Display_Table tuple_display_table_;

  // Stores the Tuples.
  Tuple_Queue tuple_queue_;
};

class Tuple
  // = TITLE
  //
  //    Common interface of all tuple types.
{
public:
  // Reads a tuple from <cin> and stores it in the <Tupleset>.
  virtual void get_tuple (Tupleset &ts, istream &is) = 0;
};

class Name_Number_Tuple : public Tuple
  // = TITLE
  //
  //    An implementation of Tuple for names and telephone numbers.
{
public:
  // = Initialization methods.
  Name_Number_Tuple (void)
    {
    }

```

```
Name_Number_Tuple (const String &lastname,
                   const String &firstname,
                   const String &telephone)
  : lastname_ (lastname),
    firstname_ (firstname),
    telephone_ (telephone)
  {
  }

// = Accessor and mutator methods.

String lastname (void) { return this→lastname_; }
String firstname (void) { return this→firstname_; }
String telephone (void) { return this→telephone_; }

void lastname (const String &l) { this→lastname_ = l; }
void firstname (const String &f) { this→firstname_ = f; }
void telephone (const String &t) { this→telephone_ = t; }

// Reads a <Tuple>. The exact strategy used is encapsulated in
// <get_tuple>.  <get_tuple> guarantees that the <Tuple> will be
// passed to <Tupleset> s after reading and that its fields contain
// the correct values.

void get_tuple (Tupleset &ts, istream &is)
  {
    char lastname[BUFSIZ];
    char firstname[BUFSIZ];
    char telephone[BUFSIZ];

    // Runs the event loop that gets the input tuple from the input
    // stream and updates the display.

    for (;;)
      {
        cout ≪ "enter lastname, firstname, phonenumber" ≪ endl
             ≪ "(separated by spaces)" ≪ endl;

        is ≫ lastname ≫ firstname ≫ telephone;

        if (is)
          {
            Name_Number_Tuple *nntp =
              new Name_Number_Tuple (lastname,
                                     firstname,
                                     telephone);
```

```
          // Insert the value of this tuple into the <Tupleset>.
          ts.new_tuple (nntp);
        }
      else
        break;
    }
  }

  String lastname_;                                  // Name of Person
  String firstname_;                                 // Firstname of Person
  String telephone_;                                 // Telephonenumber of Person
};

class Name_Number_Tuple_Display_1 : public Tuple_Display
  // = TITLE
  //
  // Displays <Name_Number_Tuple> objects in the order in which they
  // are delivered, showing all their components in a simple format.
{
public:
  Name_Number_Tuple_Display_1 (const String &name)
    : Tuple_Display (name)
    {
    }

  virtual void new_tuple (Tuple *t)
    {
      Name_Number_Tuple *nnt = (Name_Number_Tuple *) t;

      // Display this Tuple on output stream.
      String s;
      s += nnt->firstname ();
      s += String (" ");
      s += nnt->lastname ();
      s += String (", ");
      s += nnt->telephone ();
      s += String ("\n");

      this->tuple_string_vector_.enqueue_tail (s);
      this->display ();
    }

  virtual void new_tuples (Tuple_Queue &tq)
    {
      Tuple **t = 0;
```

```

      // Display all the tuples in the <Tuple_Queue>.
      for (Tuple_Queue_Iterator iter (tq);
           iter.next (t) ≠ 0;
           iter.advance ())
        this→new_tuple (∗t);
    }

  // Displays the contents of <tuple_string_vector_> on the output
  // stream.

  void display (void)
    {
      if (this→tuple_string_vector_.size () > 0)
        {
          cout ≪ "===========" ≪ endl
               ≪ "Displaying " ≪ this→display_name_ ≪ endl;

          String ∗s = 0;
          size_t count = 0;

          for (Tuple_String_Vector_Iterator iter (this→tuple_string_vector_);
               iter.next (s) ≠ 0;
               iter.advance ())
            cout ≪ count++ ≪ ":   " ≪ ∗s;

          cout ≪ "===========" ≪ endl ≪ endl;
        }
    }

private:
  // <tuple_string_vector_> contains the display presentation of the
  // <Tuples>.  The contents of <tuple_string_vector_> are written
  // into the output stream as is.
  Tuple_String_Vector tuple_string_vector_;
};

class Tuple_Display_A : public Tuple_Display
  // = TITLE
  //
  //     Tuple_Display_A displays Tuplesets in a style that can be
  //     adapted to many different purposes by inserting an appropriate
  //     selection method (reject Tuples completely), comparison method
  //     (define a Tuple order) and formatting method (select Tuple
  //     components and format them).
{
```

```
public:
  Tuple_Display_A (const String &name)
    : Tuple_Display (name)
    {
    }

  // Returns desired representation of Tuple a as a String.  The
  // String may contain zero, one, or more newline characters.

  virtual String format (Tuple *a) = 0;

  // Returns true, if the Tuple should be displayed and false otherwise.
  virtual int select (Tuple *a) = 0;

  // Returns true, if Tuple a should be displayed before Tuple b and
  // false, if Tuple b should be displayed before Tuple a.
  virtual int less_than (Tuple *a, Tuple *b) = 0;

  // Implements adding a new <Tuple>.  First <select> is used to test,
  // whether the <Tuple> should be added at all, then <merge_in> moves
  // it to the right place in the presentation using <less_than> and
  // <format> converts it into a String of the desired display format.
  //
      *** DESIGN PATTERN: ***
      new_tuple () together with its auxiliary method merge_in () forms a
      **Template Method**. The empty spots that are filled in subclasses
      are the methods select (), format (), and less_than ().

  virtual void new_tuple (Tuple *name_number)
    {
      if (this->select (name_number) == 0)
        return;                                  // This Tuple won't be displayed

      // Puts name_number into tuple_queue_ and tuple_string_vector_
      this->merge_in (name_number);
      this->display ();
    }

  // Analog to <new_tuple>, but uses a loop for adding several
  // <Tuples> at once. The actual display is updated only once at the
  // end.

  virtual void new_tuples (Tuple_Queue &tq)
    {
      Tuple **name_number = 0;

```

```
    for (Tuple_Queue_Iterator iter (tq);
         iter.next (name_number) ≠ 0;
         iter.advance ())
      {
        if (this→select (∗name_number))
          this→merge_in (∗name_number);
        else
          ;                                              // ignore Tuple
      }

    this→display ();
  }

// Finds the place in <tuple_queue_> where the new <Tuple> should be
// and inserts it there.  Generates the final display representation
// of the <Tuple> and inserts that in <tuple_string_vector_>.

void merge_in (Tuple ∗name_number)
  {
    // Index where the new <Tuple> must be inserted.
    int where;

    for (where = 0;
         where < tuple_queue_.size ();
         where++)
      {
        Tuple ∗∗tuple = 0;

        // Locate the <tuple ∗> at location <where>.
        this→tuple_queue_.get (tuple, where);

        // If <tuple> is less than <name_number> then we've found
        // where <name_number> belongs so we can exit the loop.
        if (this→less_than (∗tuple, name_number) == 0)
          break;
      }

    // Now <where> indicates the target index where <name_number>
    // belongs (in sorted order).  Move all other elements one
    // position towards the end in both <tuple_queue_> and
    // <tuple_string_vector_>.

    int size = tuple_queue_.size ();

    for (int k = size - 1;                               // Start at end.
         k ≥ where;
```

```
                k--)
            {
                Tuple **t = 0;
                tuple_queue_.get (t, k);
                tuple_queue_.set (*t, k + 1);

                String *s = 0;
                tuple_string_vector_.get (s, k);
                tuple_string_vector_.set (*s, k + 1);
            }

        // Now insert new <Tuple> into <tuple_queue_> and its formatted
        // counterpart into <tuple_string_vector_> at position <where>.

        tuple_queue_.set (name_number, where);
        tuple_string_vector_.set (this→format (name_number), where);
    }

  // Displays the contents of <tuple_string_vector_> on the output
  // stream.

  void display (void)
    {
      if (this→tuple_string_vector_.size () > 0)
        {
          cout ≪ "===========" ≪ endl
                ≪ "Displaying " ≪ this→display_name_ ≪ endl;

          String *s = 0;
          size_t count = 0;

          for (Tuple_String_Vector_Iterator iter (this→tuple_string_vector_);
                iter.next (s) ≠ 0;
                iter.advance ())
            cout ≪ count++ ≪ ":    " ≪ *s;

          cout ≪ "===========" ≪ endl ≪ endl;
        }
    }

private:
  // <tuple_queue_> contains all <Tuples> that are to be displayed
  // (after filtering!), their order represents the presentation
  // order.
  Tuple_Queue tuple_queue_;

```

```
  // <tuple_string_vector_> parallels <tuple_queue_> and contains the
  // final display presentation of the <Tuples>.  The contents of
  // <tuple_string_vector_> are written into the output stream as is.
  Tuple_String_Vector tuple_string_vector_;
};

// Main program.
     *** DESIGN PATTERN: ***
     The two Tuple_Displays are registered as observers at the Tupleset.

int
main (void)
{
  Tupleset tuple_set;

  Tuple_Display *disp1 =
    new Name_Number_Tuple_Display_1 (String ("chronologically"));

  tuple_set.new_display (disp1);

  Name_Number_Tuple nnt;

  // Runs the event loop that gets the input tuple from the input
  // stream and updates the display.
  nnt.get_tuple (tuple_set, cin);

  return 0;
}

```

B.2 Program "Element"

```
/*


####################################
This program is enhanced by:
     PUT IN YOUR NAME!!!
####################################


This program manages And/Or-Sequences of <Strings>.  Such sequences
consist of <Element> objects. There are three kinds of <Elements>:

  1. <String_Element> – which contains a single <String>.

```

```
   2. <And_Element> – which contains a list of elements that are
      concatenated in the given order.

   3. <Or_Element> – which contains a set of elements that are used
      alternatively, exactly one at a time.

The <And_Element> and the <Or_Element> objects can be nested
arbitrarily.  The lowest level (leafs) of any object tree formed this
way always contains <String_Element> objects.  <Or_Elements> result in
variants.  Where <Or_Elements> meet, the cross product of their
variants is formed.

Here is the class hierarchy:

   abstract class Element
      class String_Element
      class And_Element
      class Or_Element

   abstract class Element_Action
      class Depth

The main function generates an And/Or sequence and prints it.

Note, do not worry about memory leaks in this code.

*** DESIGN PATTERN: ***

 Element is the abstract superclass of a **Composite**.
 String_Element is its leaf type.
 And_Element and Or_Element are its container (composite) types

 Element_Action is the abstract superclass of a **Visitor**.
 The data structure that is visited is an Element container nesting.

*/

// Forward declarations.
class Element;
class String_Element;
class And_Element;
class Or_Element;

// These are necessary to provide String and Queue abstractions.

#include "ace/SString.h"
```

```
#include "ace/Containers.h"

// Some simple typedefs.
typedef ACE_CString String;
typedef ACE_Unbounded_Queue<Element *> Element_Queue;
typedef ACE_Unbounded_Queue_Iterator<Element *> Element_Queue_Iterator;

class Element_Action
  // = TITLE
  //
  //    Interface for operations on Element structures that must handle
  //    the parts differently, depending on their type.
  *** DESIGN PATTERN: ***
  Design pattern Visitor: Element_Action is the superclass of all
  visitors of 'Element' Composite data structures.
{
public:
  // Perform the action for <String_Elements>.
  virtual void string_action (String_Element &e) = 0;

  // Perform the action for <And_Elements>.
  virtual void and_action (And_Element &e) = 0;

  // Perform the action for <Or_Elements>.
  virtual void or_action (Or_Element &e) = 0;
};

class Element
  // = TITLE
  //
  //     Interface of the 'Element' classes, i.e., String_Element,
  //     And_Element, Or_Element.
  //
 DESIGN PATTERN
 Element is the superclass of a Composite pattern.
{
public:
  // Input operation. Adds another element to this element.
  virtual void add (Element *e) = 0;

  // Output operation.  Prints the whole <Element> on <cout>.  Each
  // variant is printed on a separate line.
  virtual String to_string (void) = 0;

  // Output operation.  Prints the whole <Element> on <cout>.  Each
  // variant is printed on a separate line.
```

```
    virtual void print (void)
      {
        Element_Queue eq = this→variants ();
        Element ∗∗e = 0;

        // Iterate through all the <Elements> and print each on a
        // separate line.
        for (Element_Queue_Iterator iter (eq);
             iter.next (e) ≠ 0;
             iter.advance ())
          // Print out the char ∗ representation of the string.
          cout ≪ (∗e)→to_string () ≪ endl;
      }

    // Representation operation. Returns a <Element_Queue> in which each
    // element contains the <String> representation of one variant of
    // the And/Or sequence.  Each variant appears exactly once in the
    // <Element_Queue>.
    virtual Element_Queue variants (void) = 0;

    // Branching operation. Calls that one operation from
    // <Element_Action> that is responsible for (i.e., corresponds to)
    // the present subclass.
        ∗∗∗ DESIGN PATTERN: ∗∗∗
        This is the ’double dispatch’ procedure in the visited data
        structure for the Visitor pattern.
    virtual void perform (Element_Action &a) = 0;
};

class String_Element : public Element
    // = TITLE
    //
    //     Element class that contains exactly one String.
    //
  DESIGN PATTERN:
  String_Element is the (only) leaf class in a Composite pattern.
{
public:
    // Generates a String_Element with empty contents.
    String_Element (void)
      {
      }

    // Generates a String_Element with the contents given as argument.
    String_Element (const String &s)
      : contents_ (s)
```

```
    {
    }

  // Input operation. Appends <e> to current String contents if <e> is
  // a <String_Element>.  We assume that <e> is a <String_Element>.
  virtual void add (Element *e)
    {
      // Perform a cast (oh what I wouldn't give for RTTI... ;-));
      String_Element &s = (String_Element &) e;

      // += is the append operator.
      this->contents_ += s.contents_;
    }

  // Representation operation. Returns a coded representation of the
  // element as a string of the form <contents_>.
  virtual String to_string (void)
    {
      return this->contents_;
    }

  // Returns a <Element_Queue> with exactly one element: the String
  // that the <String_Element> contains.
  virtual Element_Queue variants (void)
    {
      Element_Queue sa;
      sa.enqueue_tail (this);
      return sa;
    }

  // Calls the string_action of the Element_Action given as argument.
      *** DESIGN PATTERN: ***
      perform() is the dispatch operation for Visitor
      (Element_Action).

  virtual void perform (Element_Action &a)
    {
      a.string_action (*this);
    }

private:
  // Contents of the String_Element.
  String contents_;
};

class And_Element : public Element
```

```
  // = TITLE
  //
  //      And sequence of elements.  That means the elements must be
  //      concatenated.
  //
  *** DESIGN PATTERN: ***
 And_Element is one of the container classes of a Composite
 pattern.
{
public:
  // Generates an <And_Element> without any contents.
  And_Element (void)
    {
    }

  // Input operation. Adds <e> as new element to current sequence of
  // elements.
  virtual void add (Element ∗e)
    {
      this→elements_.enqueue_tail (e);
    }

  // Representation operation.  Returns a coded representation of the
  // Element as a String of the form ”AND ( el1 & el2 & el3 )”.
  virtual String to_string (void)
    {
      String b ("AND ( ");

      Element ∗∗e = 0;

      // Iterate through the set of <elements_> and convert them into
      // a String.

      Element_Queue_Iterator iter (this→elements_);

      while (iter.next (e) ≠ 0)
        {
          b += (∗e)→to_string ();

          // Check to see if we're at the end of the iteration.
          if (iter.advance () ≠ 0)
            b += String ("& ");
        }

      b += String (") ");

```

```
      return b;
    }

// Returns cross product of all variants of all elements in the
// <And_Element>.  The product is formed by concatenating the
// Strings of the variants.

virtual Element_Queue variants (void)
    {
      Element_Queue result;
      Element ∗∗e = 0;

      // Iterate over all the elements.

      for (Element_Queue_Iterator iter (this→elements_);
           iter.next (e) ≠ 0;
           iter.advance ())
        {
          Element_Queue nextpart ((∗e)→variants ());

          if (nextpart.size () == 0)
            continue;                          // Nothing to add to result for this element.
          if (result.size () == 0)
            {
              result = nextpart;
              continue;                        // Nothing to combine with yet.
            }

          // Store <result> in <oldreset> and reset <result> to be
          // empty in preparation for the cross product calculation.
          Element_Queue oldresult = result;
          result.reset ();

          size_t old_n = oldresult.size ();
          size_t next_n = nextpart.size ();

          // Generate cross product (by String concatenation):
          //
          // result := oldresult x nextpart

          for (size_t old_i = 0;
               old_i < old_n;
               old_i++)
            {
              Element ∗∗ith_element;
              // Get the "ith" element in <oldresult>.
```

```
                    oldresult.get (ith_element, old_i);

                    String current_old = (*ith_element)→to_string ();

                    for (size_t next_i = 0;
                         next_i < next_n;
                         next_i++)
                      {
                        // Find the ”ith” element in <nextpart>.
                        nextpart.get (ith_element, next_i);

                        String current_next = (*ith_element)→to_string ();

                        // Concatenate the strings.
                        String_Element *concat =
                          new String_Element (current_old + current_next);

                        // Insert the concatenated strings at the end of the
                        // result.
                        result.enqueue_tail (concat);
                      }
                  }
              }

          return result;
        }

    // Calls the string_action of the Element_Action given as argument.
          *** DESIGN PATTERN: ***
          perform() is the dispatch operation for Visitor
          (Element_Action).

    virtual void perform (Element_Action &a)
      {
        a.and_action (*this);
      }

    // @@ private:
    // Array of the elements forming the <And_Element>.
    Element_Queue elements_;
};

class Or_Element : public Element
  // = TITLE
  //
  //    Or set of elements.
```

```
  //     (That means the elements are alternatives.)
  //
  *** DESIGN PATTERN: ***
  Or_Element is one of the container classes of a Composite pattern.
{
public:

  // Generates an Or_Element without Contents.
  Or_Element (void)
    {
    }

  // Input operation. Adds e as new element to current set of
  // alternative elements.

  virtual void add (Element *e)
    {
      this→elements_.enqueue_tail (e);
    }

  // Representation operation. Returns a coded representation of the
  // Element as a String of the form "OR (el1 | el2 | el3 )".

  virtual String to_string (void)
    {
      String b ("OR ( ");

      Element **e = 0;

      // Iterate through the set of <elements_> and convert them into
      // a String.

      Element_Queue_Iterator iter (this→elements_);

      while (iter.next (e) ≠ 0)
        {
          b += (*e)→to_string ();

          // Advance the iterator
          if (iter.advance () ≠ 0)
            b += String ("| ");
        }

      b += String (") ");

      return b;
```

```
    }

  // Returns the union of all variants of all <elements_> in the
  // <Or_Element>.

  virtual Element_Queue variants (void)
    {
      Element_Queue result;

      Element ∗∗oe = 0;

      for (Element_Queue_Iterator outer_iter (this→elements_);
           outer_iter.next (oe) ≠ 0;
           outer_iter.advance ())
        {
          Element_Queue nextpart = (∗oe)→variants ();

          Element ∗∗ie = 0;

          for (Element_Queue_Iterator inner_iter (nextpart);
               inner_iter.next (ie) ≠ 0;
               inner_iter.advance ())
            result.enqueue_tail (∗ie);
        }

      return result;
    }

  // Calls the string_action of the Element_Action given as argument.
      ∗∗∗ DESIGN PATTERN: ∗∗∗
      perform() is the dispatch operation for Visitor
      (Element_Action).

  virtual void perform (Element_Action &a)
    {
      a.or_action (∗this);
    }

  // @@ private:
  Element_Queue elements_;                 // vector of the elements forming the Or_Element
};

class Depth : public Element_Action
  // = TITLE
  //
  //   Class for computing maximum depths of the different kinds of nodes
```

```
  //   in an And/Or Element tree.
  *** DESIGN PATTERN: ***
  Depth is a Visitor of the Composite data structure Element.
  Element.perform() is the double dispatch operation of the Visitor.
{
public:
  // Walks through Element e to initialize itself.  Components
  // max_depth, max_and_depth, and max_or_depth can be queried afterwards.

  Depth (Element *e)
    : max_depth_ (-1),
      max_and_depth_ (-1),
      max_or_depth_ (-1),
      depth_ (0)
    {
      e->perform (*this);
    }

  virtual void string_action (String_Element &)
    {
      if (this->max_depth_ < this->depth_)
        this->max_depth_ = this->depth_;
    }

  virtual void and_action (And_Element &u)
    {
      if (this->max_and_depth_ < this->depth_)
        {
          this->max_and_depth_ = this->depth_;

          if (this->max_depth_ < this->depth_)
            this->max_depth_ = this->depth_;
        }

      this->iterate (u.elements_);
    }

  virtual void or_action (Or_Element &o)
    {
      if (this->max_or_depth_ < this->depth_)
        {
          this->max_or_depth_ = this->depth_;

          if (this->max_depth_ < this->depth_)
            this->max_depth_ = this->depth_;
        }
```

```

      this->iterate (o.elements_);
    }

  // = These should be private ;-)
  int max_depth_;                                  // Depth of deepest Element
  int max_and_depth_;                              // Depth of deepest And_Element
  int max_or_depth_;                               // Depth of deepest Or_Element

private:
  // Handle all elements contained in an <And_Element> or an
  // <Or_Element>.

  void iterate (Element_Queue &elems)
    {
      this->depth_++;                              // these elements are one level deeper

      Element **e;

      for (Element_Queue_Iterator iter (elems);
           iter.next (e) != 0;
           iter.advance ())
        (*e)->perform (*this);

      this->depth_--;
    }

  int depth_;                                      // current depth (root == 0)
};

// main program. generates an And/Or sequence and prints it.

int
main (int, char *[])
{
  And_Element *u = new And_Element;
  Or_Element *modal = new Or_Element;
  Or_Element *verb = new Or_Element;

  u->add (new String_Element ("Who "));
  modal->add (new String_Element ("can "));
  modal->add (new String_Element ("will "));
  u->add (modal);
  verb->add (new String_Element ("think "));
  verb->add (new String_Element ("act "));
  u->add (verb);
```

```
  u→add (new String_Element ("after all"));
  u→add (new String_Element ("?"));

  u→print ();

  cout ≪ u→to_string ();

  Depth t (u);
  cout ≪ endl
       ≪ " Depth = " ≪ t.max_depth_
       ≪ " AndDepth = " ≪ t.max_and_depth_
       ≪ " OrDepth = " ≪ t.max_or_depth_ ≪ endl;
  return 0;
}
```

Bibliography

[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] Norman E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, London, 1991.

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

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

[8] 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.

[9] Lutz Prechelt, Barbara Unger, and Michael Philippsen. Documenting design patterns in code eases program maintenance. In *Proc. ICSE Workshop on Process Modeling and Empirical Studies of Software Evolution*, pages 72–76, Boston, MA, May 1997.

[10] 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").

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